Designed for Microsoft® Windows NT® Windows®98

CD-ROM Included

Microsoft® 程序设计系列

完全修订和更新!

Windows® 程序设计

（第5版）上册

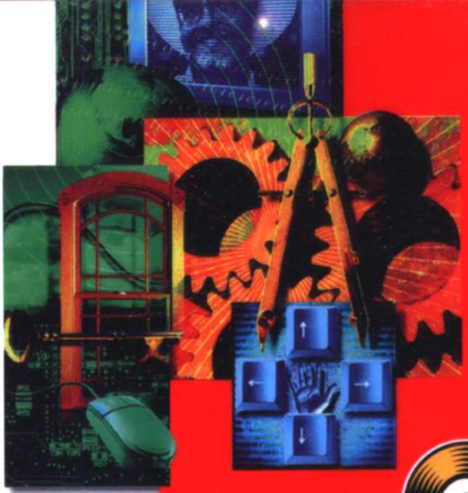
Win32 API 的权威指导

北京大学出版社

Microsoft Press

Windows® 程序设计

（第 5 版）上册



全球最著名、获得赞誉最多、使用最广泛的程序设计图书

如果对 Windows 开发有什么问题的话，请翻阅 Petzold 的著作吧。在《Windows 程序设计》（第 5 版）中，这位杰出的"Windows 先锋奖"(Windows Pioneer Award)获得者根据最新的 Windows 操作系统权威技术修订了他的经典著作——再一次演示了基本的 Win32 程序设计的 API 核心内容。本书的内容包括:

- 基本内容——输入、输出和对话框
- Unicode 概述
- 图形——绘图、文本和字体、位图和元文件
- 内核和打印机
- 声音和音乐
- 动态链接库
- 多任务和多线程
- 多文档界面
- Internet 和 Intranet 的程序设计

其中采用的大多是具有代表性的示例，这本 Petzold 著作为使用 Windows 95、Windows 98 或 Windows NT 的各级 Windows 程序员提供了最基本的参考和指导。没有经验的开发人员也可以从中获取大量的新知识。

配套光盘包括：

- 所有代表性应用程序的源代码和 EXE 文件
- 本书完整的 HTML 版本

系统要求

要使用光盘中的示例程序，您的系统必须运行 Windows 98、Windows NT 4.0 或更高版本。

作者介绍

Charles Petzold 是由《Windows Magazine》和 Microsoft 公司评选的 7 名"Windows 先锋奖"获得者之一，并且是唯一的一名作家。作为作者和贡献者，他在 Windows 程序设计领域有独特的见解。目前他居住在纽约。

> *"关于 Windows 程序设计最具权威性的图书当然是 Charles Petzold 的著作。"*
>
> *Dr. Dobb's Journal*

北京大学出版社

Microsoft Press

Windows 程序设计

（第 5 版）

上　册

[美]Charles Petzold　著

北京博彦科技发展有限公司　译

北 京 大 学 出 版 社

·北　京·

著作权合同登记　图字：01-1999-1925 号

Programming Windows (Fifth Edition)

Charies Petzold

本书版权为 Charles Petzold 所有，1999。(Copyright © 1999 by Charles Petzold. All rights reserved.)

本书中文版由美国 Microsoft 出版社授权北京大学出版社独家出版，1999。

本书封面贴有北京大学出版社的激光防伪标签，无标签者不得销售。

版权所有，翻印必究。

图书在版编目(CIP)数据

Windows 程序设计：(第 5 版)/[美] Charles Petzold 著；北京博彦科技发展有限公司译.—北京：北京大学出版社，1999.11

ISBN 7-301-04187-X

Ⅰ.W…　Ⅱ.①C…　②北…　Ⅲ.窗口软件，Windows 程序设计　Ⅳ.TP316

中国版本图书馆 CIP 数据核字(1999)第 31642 号

书　　名：Windows 程序设计(第 5 版)

著作责任者：[美] Charles Petzold　著　北京博彦科技发展有限公司　译

责 任 编 辑：邱淑清　沈承凤　柯　纪　张　华

标 准 书 号：ISBN 7-301-04187-X/TP·463

出　版　者：北京大学出版社

地　　　址：北京市海淀区中关村北京大学校内　100871

网　　　址：http://cbs.pku.edu.cn

电　　　话：出版部 62752015　发行部 62754140　编辑室 62757065

电 子 信 箱：zpup@pup.pku.edu.cn

排　印　者：北京大学印刷厂

发　行　者：北京大学出版社

经　销　者：新华书店

787 毫米×1092 毫米　16 开本　86 印张　1922 千字

1999 年 11 月第 1 版　2003 年 6 月第 6 次印刷

印　　　数：23001～26000 册

定　　　价：160.00 元(上下两册，含光盘)

对《Windows 程序设计》的评论

“第一流的 Windows 程序设计的书籍当然是 Charles Petzold 的《Windows 程序设计》。如果您还没有一本，您就需要得到一本。”

Windows Tech Journal

“重要的程序员指南……现在是并且继续是 Charles Petzold 的《Windows 程序设计》。如果您想要编写 Windows 应用程序并且还没有读过这本书，您就需要好好地读一读它。”

IEEE Micro

“我认为仅有三本书对于 Windows 程序设计是至关重要的：第一本，当然是由 Charles Petzold 著的经典的《Windows 程序设计》。”

INFOWORLD

“当您收藏最好的 Windows 书籍时，应该确认包括了经典的——Petzold 的最畅销的《Windows 程序设计》……”

PC Techniques

“它一直是经典的 Windows 程序设计指南。”

Programmer's Journal

“如果您想要编写 Windows 程序，请买这本书。随着时间的流逝，它会给您带来比书的价钱多得多的利益。”

Computer Language

“包含的范围很广而遗漏的很少，这本书对于认真对待 Windows 的任何人都是必需的。”

Byte Magazine

作者的话

关于此书的更新信息请访问 Web 站点：www.cpetzold.com，其中包括了可能存在的错误报告和新的代码清单。如果您对本书有什么问题，可以发电子邮件给 charles@cpetzold.com。虽然我会尽量回答您问的任何简单的问题，但是我不能做任何保证。

我要感谢微软出版社中使本书能够出版而做出大量工作的每一个人。我认为《Windows 程序设计》的"十周年版本"* 是最好的一个版本。Microsoft 的许多人（包括 Microsoft Windows 的早期开发者）也在我写前几版时，帮助我解决了很多难题，这些杰出的人们都已列在那些版本中。

也感谢我的家人和朋友，尤其是那些新朋友，是这些朋友的支持使这本书得以问世。我真诚地将此书献给你们。

Charles Petzold

1998 年 10 月 5 日

---

* 即指这本书：《Windows 程序设计》（第 5 版）。——编者注

作者简介

Charles Petzold 从 1984 年开始编写个人计算机程序，从 1985 年开始编写 Microsoft Windows 程序。他在《Microsoft Systems Journal》1986 年 12 月号上发表了第一篇关于 Windows 程序设计的杂志论文。从 1986 年到 1995 年，他为《PC Magazine》撰写“Environments”专栏，给读者介绍了 Windows 和 OS/2 程序设计的许多方面。

《Windows 程序设计》由微软出版社在 1988 年首次出版，后来被认为是这方面最好的导论性读本。在 1994 年 5 月，Petzold 作为仅有的七个人之一（并且是唯一的作家）被《Windows Magazine》和 Microsoft 公司授予 Windows Pioneer 奖，以表彰他对 Microsoft Windows 的成功做出的贡献。

在 1999 年秋天，微软出版社将出版 Charles Petzold 面向普通读者的第一本书。暂时命名为《Code: The Hidden Language of Computer Hardware and Software》，这本书专门介绍数字信息的本质以及计算机处理数字信息的方式。

目录

对《Windows 程序设计》的评论 ······ *1*
作者的话 ······ *3*
作者简介 ······ *5*

上　　册

第一部分　基础知识

第一章　开始 ······ 1
1.1　Windows 环境 ······ 2
1.1.1　Windows 简史 ······ 2
1.1.2　Windows 方面 ······ 3
1.1.3　动态链接 ······ 5
1.2　Windows 编程选项 ······ 6
1.2.1　API 和内存模式 ······ 6
1.2.2　语言选项 ······ 7
1.2.3　编程环境 ······ 8
1.2.4　API 文档 ······ 9
1.3　编写一个 Windows 程序 ······ 9
1.3.1　字符模式(Character-Mode)模型 ······ 9
1.3.2　Windows 等价程序 ······ 10
1.3.3　头文件 ······ 11
1.3.4　程序入口点 ······ 12
1.3.5　MessageBox 函数 ······ 13
1.3.6　编译、链接和运行 ······ 14

第二章 Unicode 简介 …… 17
2.1 字符集简史 …… 17
2.1.1 美国标准 …… 18
2.1.2 国际方面 …… 19
2.1.3 扩展 ASCII …… 20
2.1.4 双字节字符集 …… 21
2.1.5 Unicode 解决方案 …… 22
2.2 宽字符和 C …… 23
2.2.1 char 数据类型 …… 23
2.2.2 宽字符 …… 24
2.2.3 宽字符库函数 …… 25
2.2.4 维护单一源代码 …… 26
2.3 宽字符和 Windows …… 28
2.3.1 Windows 头文件类型 …… 28
2.3.2 Windows 函数调用 …… 30
2.3.3 Windows 的字符串函数 …… 31
2.3.4 在 Windows 中使用 printf …… 31
2.3.5 格式化消息框 …… 33
2.3.6 本书与国际化 …… 35

第三章 窗口和消息 …… 37
3.1 自己的窗口 …… 37
3.1.1 总体结构 …… 37
3.1.2 HELLOWIN 程序 …… 39
3.1.3 总体考虑 …… 42
3.1.4 注册窗口类 …… 46
3.1.5 创建窗口 …… 52
3.1.6 显示窗口 …… 53
3.1.7 消息循环 …… 54
3.1.8 窗口过程 …… 56
3.1.9 处理消息 …… 56
3.1.10 播放声音文件 …… 57
3.1.11 WM_PAINT 消息 …… 58
3.1.12 WM_DESTROY 消息 …… 60
3.2 Windows 编程的难点 …… 60
3.2.1 别调用我,我会调用您 …… 60

3.2.2 进队消息与不进队消息 …… 62
3.2.3 行动迅速 …… 63

第四章 输出文本 …… 65
4.1 绘制和刷新 …… 66
4.1.1 WM_PAINT 消息 …… 66
4.1.2 有效矩形和无效矩形 …… 67
4.2 GDI 简介 …… 68
4.2.1 设备描述表 …… 68
4.2.2 获取设备描述表句柄：方法一 …… 69
4.2.3 绘图信息结构 …… 70
4.2.4 获取设备描述表句柄：方法二 …… 72
4.2.5 TextOut：细节 …… 72
4.2.6 系统字体 …… 74
4.2.7 字符大小 …… 74
4.2.8 文本尺寸：细节 …… 75
4.2.9 格式化文本 …… 77
4.2.10 综合使用 …… 78
4.2.11 SYSMENTS1.C 窗口过程 …… 85
4.2.12 空间不够 …… 86
4.2.13 客户区的大小 …… 87
4.3 滚动条 …… 88
4.3.1 滚动条的范围和位置 …… 89
4.3.2 滚动条消息 …… 91
4.3.3 在 SYSMETS 中添加滚动功能 …… 93
4.3.4 绘图程序的组织 …… 97
4.4 建立更好的滚动 …… 98
4.4.1 滚动条信息函数 …… 98
4.4.2 滚动范围 …… 100
4.4.3 新 SYSMETS …… 101
4.4.4 不用鼠标怎么办 …… 108

第五章 图形基础 …… 109
5.1 GDI 的结构 …… 109
5.1.1 GDI 原理 …… 109
5.1.2 GDI 函数调用 …… 111
5.1.3 GDI 图元 …… 112
5.1.4 其他方面 …… 112

5.2 设备描述表 …… 113
5.2.1 获取设备描述表句柄 …… 113
5.2.2 获取设备描述表信息 …… 115
5.2.3 DEVCAPS1 程序 …… 116
5.2.4 设备的大小 …… 119
5.2.5 关于色彩 …… 124
5.2.6 设备描述表属性 …… 126
5.2.7 保存设备描述表 …… 127
5.3 画点和线 …… 128
5.3.1 写像素 …… 129
5.3.2 直线 …… 129
5.3.3 边界框函数 …… 134
5.3.4 贝塞尔样条 …… 140
5.3.5 使用现有画笔(Stock Pens) …… 145
5.3.6 画笔的创建、选择和删除 …… 146
5.3.7 填充空隙 …… 150
5.3.8 绘图方式 …… 150
5.4 绘制填充区域 …… 152
5.4.1 Polygon 函数和多边形填充方式 …… 153
5.4.2 用画刷填充内部 …… 157
5.5 GDI 映射方式 …… 159
5.5.1 设备坐标和逻辑坐标 …… 161
5.5.2 设备坐标系 …… 161
5.5.3 视口和窗口 …… 162
5.5.4 处理 MM_TEXT …… 164
5.5.5 "度量"映射方式 …… 166
5.5.6 "自作主张的"映射方式 …… 169
5.5.7 WHATSIZE 程序 …… 175
5.6 矩形、区域和剪裁 …… 178
5.6.1 矩形函数 …… 178
5.6.2 随机矩形 …… 180
5.6.3 创建和绘制区域 …… 184
5.6.4 矩形与区域的剪裁 …… 186
5.6.5 CLOVER 程序 …… 187

第六章 键盘 …… 193
6.1 键盘基础 …… 193
6.1.1 忽略键盘 …… 193

6.1.2 谁获得了焦点 …… 194
6.1.3 队列和同步 …… 195
6.1.4 击键和字符 …… 195
6.2 击键消息 …… 196
6.2.1 系统击键与非系统击键 …… 196
6.2.2 虚拟键码 …… 197
6.2.3 lParam 信息 …… 201
6.2.4 换档状态 …… 202
6.2.5 使用击键消息 …… 203
6.2.6 为键盘增强 SYSMETS …… 204
6.3 字符消息 …… 212
6.3.1 四类字符消息 …… 212
6.3.2 消息顺序 …… 213
6.3.3 处理控制字符 …… 215
6.3.4 死字符消息 …… 216
6.4 键盘消息和字符集 …… 216
6.4.1 KEYVIEW1 程序 …… 217
6.4.2 外语键盘问题 …… 222
6.4.3 字符集和字体 …… 224
6.4.4 Unicode 怎么样? …… 234
6.4.5 TrueType 和大字体 …… 235
6.5 插入符(不是光标) …… 242
6.5.1 插入符函数 …… 242
6.5.2 TYPER 程序 …… 243

第七章 鼠标 …… 251
7.1 鼠标基础 …… 251
7.1.1 一些简单的定义 …… 252
7.2 客户区鼠标消息 …… 253
7.2.1 简单的鼠标处理：一个例子 …… 254
7.2.2 处理 Shift 键 …… 258
7.2.3 双击鼠标键 …… 259
7.3 非客户区鼠标消息 …… 261
7.3.1 命中测试消息 …… 262
7.3.2 从消息产生消息 …… 263
7.4 程序中的命中测试 …… 263
7.4.1 一个假想的例子 …… 264
7.4.2 示例程序 …… 264

7.4.3　使用键盘仿真鼠标 …… 268
7.4.4　在 CHECKER 中添加键盘接口 …… 269
7.4.5　将子窗口用于命中测试 …… 273
7.4.6　CHECKER 中的子窗口 …… 274
7.4.7　子窗口和键盘 …… 279
7.5　捕获鼠标 …… 284
7.5.1　设计矩形 …… 284
7.5.2　捕获的解决方案 …… 289
7.5.3　BLOKOUT2 程序 …… 289
7.6　鼠标轮 …… 293
7.6.1　后会有期 …… 301

第八章　计时器 …… 303
8.1　计时器基础 …… 304
8.1.1　系统和计时器 …… 304
8.1.2　计时器消息不是异步的 …… 305
8.2　计时器的使用：三种方法 …… 305
8.2.1　方法一 …… 305
8.2.2　方法二 …… 309
8.2.3　方法三 …… 312
8.3　计时器用于时钟 …… 313
8.3.1　构造数字时钟 …… 313
8.3.2　获取当前时间 …… 318
8.3.3　显示数字和冒号 …… 319
8.3.4　国际化 …… 319
8.3.5　构造模拟时钟 …… 320
8.4　为状态报告使用计时器 …… 326

第九章　子窗口控制 …… 331
9.1　按钮类 …… 333
9.1.1　创建子窗口 …… 337
9.1.2　子窗口向父窗口发送消息 …… 338
9.1.3　父窗口向子窗口发送消息 …… 339
9.1.4　下压按钮 …… 340
9.1.5　复选框 …… 341
9.1.6　单选按钮 …… 342
9.1.7　分组框 …… 342
9.1.8　更改按钮文本 …… 343

9.1.9 可见的和启用的按钮 …… 343
9.1.10 按钮和输入焦点 …… 344
9.2 控制与颜色 …… 345
9.2.1 系统颜色 …… 345
9.2.2 按钮颜色 …… 347
9.2.3 WM_CTLCOLORBTN 消息 …… 348
9.2.4 拥有者绘制按钮 …… 348
9.3 静态类 …… 355
9.4 滚动条类 …… 356
9.4.1 COLORS1 程序 …… 357
9.4.2 自动键盘接口 …… 364
9.4.3 窗口子类化 …… 365
9.4.4 给背景着色 …… 366
9.4.5 给滚动条和静态文本着色 …… 367
9.5 编辑类 …… 368
9.5.1 编辑类风格 …… 370
9.5.2 编辑控制通知 …… 371
9.5.3 使用编辑控制 …… 372
9.5.4 发送给编辑控制的消息 …… 372
9.6 列表框类 …… 373
9.6.1 列表框风格 …… 374
9.6.2 将串放入列表框 …… 374
9.6.3 选择和获取项 …… 376
9.6.4 接收来自列表框的消息 …… 377
9.6.5 一个简单的列表框应用程序 …… 378
9.6.6 文件列表 …… 382
9.6.7 Windows 的 head 程序 …… 383

第十章 菜单及其他资源 …… 389
10.1 图标、光标、字符串和定制资源 …… 390
10.1.1 将图标添加到程序 …… 390
10.1.2 获取图标句柄 …… 395
10.1.3 在程序中使用图标 …… 398
10.1.4 使用自定义光标 …… 399
10.1.5 字符串资源 …… 399
10.1.6 定制资源 …… 401
10.2 菜单 …… 409
10.2.1 菜单概念 …… 410

10.2.2 菜单结构 …… 410
10.2.3 定义菜单 …… 411
10.2.4 在程序中引用菜单 …… 411
10.2.5 菜单和消息 …… 412
10.2.6 示例程序 …… 414
10.2.7 菜单设计规范 …… 421
10.2.8 定义菜单的复杂方法 …… 421
10.2.9 浮动弹出式菜单 …… 423
10.2.10 使用系统菜单 …… 428
10.2.11 更改菜单 …… 431
10.2.12 其他菜单命令 …… 432
10.2.13 创建菜单的非正规方法 …… 433
10.3 键盘加速键 …… 438
10.3.1 为什么要使用加速键 …… 438
10.3.2 安排加速键的几条规则 …… 439
10.3.3 加速键表 …… 439
10.3.4 加载加速键表 …… 440
10.3.5 键盘代码转换 …… 440
10.3.6 接收加速键消息 …… 441
10.3.7 菜单与加速键应用程序 POPPAD …… 442
10.3.8 启用菜单项 …… 448
10.3.9 处理菜单项 …… 449

第十一章 对话框 …… 453
11.1 模态对话框 …… 454
11.1.1 创建 About 对话框 …… 454
11.1.2 对话框及其模板 …… 458
11.1.3 对话框过程 …… 461
11.1.4 激活对话框 …… 462
11.1.5 不同的主题 …… 463
11.1.6 更复杂的对话框 …… 466
11.1.7 使用对话框控制 …… 473
11.1.8 OK 和 Cancel 按钮 …… 476
11.1.9 避免全局变量 …… 477
11.1.10 Tab 停留位和组 …… 478
11.1.11 在对话框上绘图 …… 480
11.1.12 将其他函数用于对话框 …… 481
11.1.13 定义自己的控制 …… 481

11.2 非模态对话框 …… 488
11.2.1 模态对话框与非模态对话框的区别 …… 489
11.2.2 新的 COLORS 程序 …… 491
11.2.3 HEXCALC：窗口还是对话框？ …… 496
11.3 通用对话框 …… 504
11.3.1 增强 POPPAD …… 504
11.3.2 Unicode 文件 I/O …… 527
11.3.3 更改字体 …… 528
11.3.4 查找与替换 …… 528
11.3.5 只调用一个函数的 Windows 程序 …… 529

第十二章 剪贴板 …… 531
12.1 剪贴板的简单使用 …… 531
12.1.1 标准剪贴板数据格式 …… 532
12.1.2 内存分配 …… 533
12.1.3 将文本传送到剪贴板 …… 535
12.1.4 从剪贴板上获取文本 …… 536
12.1.5 打开和关闭剪贴板 …… 537
12.1.6 剪贴板和 Unicode …… 538
12.2 复杂的剪贴板用法 …… 544
12.2.1 利用多个数据项 …… 544
12.2.2 延迟生成 …… 546
12.2.3 私有数据格式 …… 547
12.3 实现剪贴板查看器 …… 549
12.3.1 剪贴板查看器链 …… 550
12.3.2 剪贴板查看器的函数和消息 …… 550
12.3.3 一个简单的剪贴板查看器 …… 553

第一部分 基础知识

第一章

开始

本书介绍了在 Microsoft Windows 98、Microsoft Windows NT 4.0 和 Windows NT 5.0 下编写程序的方法。这些程序用 C 语言编写并使用原始的 Windows 应用程序编程接口(API)。正如在本章稍后所讨论的,这不是编写 Windows 程序的唯一方法。然而,无论最终您使用什么方式编写代码,了解 Windows API 都是非常重要的。

正如您可能知道的,Windows 98 是最新的图形操作系统的代表,已成为围绕 32 位 Intel 微处理器(例如 486 和 Pentium)的 IBM 兼容的个人计算机事实上的标准。Windows NT 是运行于 PC 兼容机及一些 RISC(精简指令集计算机)工作站的 Windows 的工业增强型版本。

使用本书有三个先决条件。首先,您应该从用户的角度熟悉 Windows 98。您不要期望不了解 Windows 用户界面就可编写其应用程序。基于此,我建议您在开发程序(或在进行其他工作)时,使用基于 Windows 的机器,并使用 Windows 应用程序。

第二,您应了解 C 语言。如果您不了解 C 语言,一开始就编写 Windows 程序并不是一个好主意。我建议您在字符环境中,例如在 Windows 98 MS-DOS 命令提示窗口下提供的环境中学习 C 语言。Windows 编程有时包括一些非字符编程方面的 C 语言部分;在这些情况下,我将提供一些针对性的讨论。但大多数情况下,您应非常熟悉该语言,特别是 C 语言的结构和指针。了解标准 C 语言运行时库的某些知识是有帮助的,但不用必须了解。

第三,您应在机器上安装一个适于进行 Windows 编程的 32 位 C 语言编译器和开发环境。在本书中,假定您正在使用 Microsoft Visual C++ 6.0,该软件包可独立购买,也可作为 Visual Studio 6.0 软件包的一部分购买。

到此为止,我将不再假设您已经具有任何图形用户界面(例如 Windows 等)的编程经验。

1.1 Windows 环境

Windows 几乎不需要介绍。然而人们很容易忘记 Windows 给办公室和家庭桌面计算机带来的巨大变化。Windows 在其早期走过了坎坷的道路,征服桌面市场的前途曾一度非常渺茫。

1.1.1 Windows 简史

在 1981 年秋天 IBM PC 推出之后不久,就已经很明显,PC 上的主流操作系统将是 MS-DOS。MS-DOS 代表 Microsoft Disk Operating System(Microsoft 磁盘操作系统)。MS-DOS 是一个小型的操作系统。MS-DOS 提供给用户一种命令行界面,以诸如 DIR 和 TYPE 这样的命令为特征,它可以将应用程序装入内存来执行。对于应用程序员,它提供了一组函数调用进行文件的输入输出(I/O)。对于其他的任务——尤其是将文本或图形写到视频显示器——应用程序可以直接访问 PC 的硬件。

由于内存和硬件的限制,成熟的图形环境是慢慢来到小型计算机中的。当苹果计算机公司那不幸的 Lisa 在 1983 年 1 月发布时,它提供了不同于字符模式环境的另一种可能的选择,并在 1984 年 1 月成为 Macintosh 上图形环境的一种标准。尽管 Macintosh 的市场占有率在下降,但是它仍然被认为是衡量所有其他图形环境的标准。包括 Macintosh 和 Windows 的所有图形环境都要归功于 Xerox Palo Alto Research Center(PARC)在 70 年代中期所做的开拓性的工作。

Windows 是由 Microsoft 在 1983 年 11 月(在 Lisa 之后 Macintosh 之前)宣布,并在两年后的 1985 年 11 月推出。在此后的两年中,紧随着 Microsoft Windows 早期版本 1.0 之后,又进行了几次更新,以支持国际商业市场,并提供新型视频显示器和打印机的驱动程序。

Windows 2.0 是在 1987 年 11 月推出的。该版本对用户界面做了一些改变。这些改变中最有效的是使用了层叠式窗口,而 Windows 1.0 中使用的是"平铺"窗口。Windows 2.0 还增强了键盘和鼠标接口,特别是加入了菜单和对话框。

至此,Windows 还只要求 Intel 8086 或者 8088 微处理器,以"实模式"运行,访问1 MB 的内存。Windows/386(在 Windows 2.0 之后不久推出的)使用 Intel 386 微处理器的"虚拟 86"模式,实现直接访问硬件的多个 MS-DOS 程序的窗口化和多任务化。为了统一起见,Windows 2.1 被更名为 Windows/286。

Windows 3.0 是在 1990 年 5 月 22 日发布的。它将 Windows/286 和 Windows/386 结合到一种产品中。Windows 3.0 有了一个很大的改变,这就是对 Intel 的 286、386 和 486 微处理器保护模式的支持。这能使 Windows 和 Windows 应用程序访问高达 16 MB 的内存。Windows 用于运行程序和维护文件的"外壳"程序得到了全面的改进。Windows 3.0

是第一个进入许多用户家庭和办公室的版本。

任何 Windows 的历史都必须包括 OS/2 的一点介绍，这是对 DOS 和 Windows 的另一种选择，最初是由 Microsoft 和 IBM 合作开发的。OS/2 版本 1.0（只有字符模式）在 Intel 286（或者后来的）微处理器上运行，在 1987 年末推出。在 1988 年 10 月的 OS/2 版本 1.1 中出现了图形的显示管理器（PM：Presentation Manager）。PM 最初的设想是成为 Windows 的一种保护模式版本，但是图形 API 改变程度太大，以致使软件生产商很难提供对这两种平台的支持。

到 1990 年 9 月，IBM 和 Microsoft 之间的冲突达到了顶峰，导致了这两个公司最终的分道扬镳。IBM 接管了 OS/2，而 Microsoft 明确表示 Windows 将是他们操作系统策略的中心。虽然 OS/2 仍然拥有一些狂热的崇拜者，但是它远未达到 Windows 这样的普及程度。

Microsoft Windows 3.1 是 1992 年 4 月推出的，其中包括的几个重要特性分别是 TrueType 字体技术（给 Windows 带来可缩放的轮廓字体）、多媒体（声音和音乐）、对象链接和嵌入（OLE：Object Linking and Embedding）和通用对话框。同样，Windows 3.1 只能在保护模式下运行，并且要求至少配置了 1 MB 内存的 286 或 386 处理器。

在 1993 年 7 月引入的 Windows NT 是支持 Intel 386、486 和 Pentium 微处理器的 32 位模式的第一个 Windows 版本。Windows NT 有 32 位的平面地址空间，并使用 32 位的指令集。（本章后面我会谈到一点地址空间。）Windows NT 还可以被移植到非 Intel 处理器上，并在基于几种 RISC 芯片的工作站上运行。

Windows 95 是在 1995 年 8 月发布的。和 Windows NT 一样，Windows 95 也支持 Intel 386 或更高级处理器的 32 位编程模式。虽然它缺少 Windows NT 中的某些功能特性，如高安全性和对 RISC 机器的可移植性等，但是 Windows 95 具有需要较少硬件资源的优点。

Windows 98 在 1998 年 6 月发布，具有许多扩展，其中包括性能的提高、更好的硬件支持，以及与 Internet 和 World Wide Web 更紧密的结合。

1.1.2 Windows 方面

Windows 98 和 Windows NT 都是 32 位抢占式多任务及多线程图形操作系统。Windows 拥有一个图形用户界面（GUI），有时也称作“可视化界面”或“图形窗口环境”。有关 GUI 的概念可追溯至 70 年代中期，在 Alto 和 Star 等机器上以及 SmallTalk 等环境中由 Xerox PARC 所做的工作。该项工作后来被苹果计算机公司和微软公司引入主流并流行起来。虽然曾一度引起争议，但现在已非常清楚，GUI 是个人计算机界唯一最重要的“交响乐”（Microsoft 的 Charles Simonyi 语）。

所有 GUI 都在基于位图的视频显示器上使用图形。图形提供了使用屏幕的最佳方式、传递信息的可视化环境、能够 WYSIWYG（what you see is what you get：所见即所得）的

图形视频显示，以及为打印文档准备好格式化文本。

在早期，视频显示器仅用于响应用户通过键盘输入的文本。在图形用户界面中，视频显示器自身成为用户输入的一个来源。视频显示器以图标和输入设备（例如按钮和滚动条）的形式显示多种图形对象。用户可以使用键盘（或者更直接地用鼠标等定位设备）直接在屏幕上操纵这些对象。可以拖动图形对象、按下鼠标按钮，以及滚动滚动条。

因此，用户与程序的交往变得更为密切。这不再是一种信息从键盘到程序，再到视频显示器的单向流动，用户已经能够与显示器上的对象直接交互。

用户不再需要花费长时间学习如何使用计算机或掌握新程序了。Windows 帮助进行这一切，因为所有应用程序都有相同的基本外观和感觉。程序占据一个窗口——屏幕上的一块矩形区域。每个窗口由一个标题栏标识。大多数程序功能由程序的菜单开始。用户可使用滚动条观察那些无法在一个屏幕中装下的信息。某些菜单项触发对话框，用户可在其中输入额外的信息。几乎在每个大的 Windows 程序中都有一个用于打开文件的特殊对话框。该对话框在所有这些 Windows 程序中看起来都一样（或基本相同），而且几乎总是从同一菜单选项中被激活。

一旦了解使用一个 Windows 程序的方法，学习其他程序就非常容易。菜单和对话框允许用户试验一个新程序并探究它的功能。大多数 Windows 程序同时具有键盘界面和鼠标界面。虽然 Windows 程序的大多数功能可通过键盘控制，但使用鼠标要容易得多。

从程序员的角度看，一致的用户界面来自于 Windows 构造菜单和对话框的内置程序。所有菜单都有同样的键盘和鼠标界面，因为这项工作是由 Windows 处理的，而不是由应用程序处理的。

为便于多个程序的使用，以及这些程序间信息的交换，Windows 支持多任务。在同一时刻能有多个 Windows 程序显示并运行。每个程序在屏幕上占据一个窗口。用户可在屏幕上移动窗口，改变它们的大小，在不同程序间切换，以及从一个程序向另一个程序传送数据。因为这些窗口看起来有些像桌面上的纸（当然，这是计算机还未占据办公桌之前的年代），Windows 有时被称作一个显示多个程序的“形象化的桌面”。

Windows 的早期版本使用一个被称为“非抢占式”多任务系统。这意味着 Windows 不使用系统定时器将处理时间分配给系统中运行的多个应用程序。程序必须自愿放弃控制以便运行其他程序。在 Windows NT 和 Windows 98 中，多任务是抢占式的，而且程序自身可分割成似乎是并发运行的多个可执行线程。

操作系统不对内存进行管理便无法实现多任务。当新程序启动、旧程序终止时，内存会出现碎片。系统必须能够将空闲的内存空间组织在一起。这要求系统移动内存中的代码和数据块。

即使是运行 8088 微处理器的 Windows 1.0 也能进行这类内存管理。在实模式限制下，这种能力仅被认为是软件工程的一个令人惊讶的成就。在 Windows 1.0 中，PC 体系

结构的640 KB内存限制在不要求任何额外内存的情况下被有效地扩展了。但 Microsoft 并未就此停步：Windows 2.0 允许 Windows 应用程序访问扩充内存(EMS)；Windows 3.0 运行于保护模式下，允许 Windows 应用程序访问多达16 MB的扩展内存。Windows NT 和 Windows 98 通过成熟的 32 位操作系统及平面存储空间摆脱了这些旧的限制。

运行于 Windows 上的程序可共享位于被称为"动态链接库"的其他文件中的例程。Windows 包括一个机制，能够在运行时用动态链接库中例程的链接程序。Windows 自身基本上是一个动态链接库集。

Windows 是一个图形界面，Windows 程序能够在视频显示器和打印机上充分利用图形和格式化文本。图形界面不仅在外观上更具有吸引力，而且还能够给用户传递高层次的信息。

为 Windows 编写的程序不直接访问屏幕和打印机等图形显示设备硬件。相反，Windows 包括了一个图形编程语言(称作图形设备接口或 GDI)，它使得显示图形和格式化文本更加容易。Windows 虚拟化了显示硬件。为 Windows 编写的程序可运行于任何具有 Windows 设备驱动程序的视频卡或打印机上。程序无需确定系统相连的设备类型。

对 Windows 开发者来说，将与设备无关的图形界面输出到 IBM PC 上并不是件轻松的事。PC 的设计是基于开放式体系结构的原则，第三方硬件制造商被鼓励为 PC 开发外围设备，而且开发了大量这样的设备。虽然出现了多种标准，PC 上的传统 MS-DOS 程序仍不得不各自支持许多不同的硬件配置。这对 MS-DOS 字处理软件来说非常普遍，它们连同 1 到 2 张有许多小文件的磁盘一同销售，每个文件支持一种指定的打印机。Windows 程序不要求这些驱动程序，因为这种支持是 Windows 的一部分。

1.1.3 动态链接

Windows 运转的核心是一个被称作"动态链接"的概念。Windows 提供了应用程序可利用的丰富的函数调用，大多数用于实现其用户界面和在视频显示器上显示文本和图形。这些函数采用动态链接库即 DLL 实现。这些动态链接库是一些具有 .DLL 扩展名或者有时是 .EXE 扩展名的文件，在 Windows 98 中通常位于\WINDOWS\SYSTEM 子目录中，在 Windows NT 中通常位于\WINNT\SYSTEM 和\WINNT\SYSTEM32 子目录中。

在早期，Windows 的主要部分只需在三个动态链接库实现。这代表了 Windows 的三个主要子系统，它们被称作 Kernel、User 和 GDI。当子系统的数目在 Windows 的最近版本中增多时，大多数典型的 Windows 程序产生的函数调用仍通过这三个模块之一。Kernel(当前由 16 位的 KRNL386.EXE 和 32 位的 KERNEL32.DLL 实现)处理所有的在传统上由操作系统内核处理的事务——内存管理、文件 I/O 和任务处理。User(由 16 位的 USER.EXE 和 32 位的 USER32.DLL 实现)指用户界面，实现所有窗口逻辑。GDI(由 16

位的 GDI.EXE 和 32 位的 GDI32.DLL 实现)是一个图形设备接口,允许程序在屏幕和打印机上显示文本和图形。

Windows 98 支持几千种应用程序可使用的函数调用。每个函数都有一个描述性的名称,如 CreateWindow 等。该函数(如您所猜想的)为程序创建窗口。所有应用程序可能使用的 Windows 函数都在头文件里声明。

在 Windows 程序中,使用 Windows 函数的方式通常与使用如 strlen 等 C 库函数的方式相同。主要的区别在于 C 库函数的机器代码链接到您的程序代码中,而 Windows 函数的代码位于您的程序之外的 DLL 中。

在运行 Windows 程序时,它通过一个被称作“动态链接”的进程与 Windows 相接。一个 Windows 的.EXE 文件拥有它使用的不同动态链接库的引用,所使用的函数即在那里。当 Windows 程序被加载到内存中时,程序中的调用被指向 DLL 函数的入口,如果该 DLL 不在内存中,系统就将其加载到内存中。

当链接 Windows 程序以产生一个可执行文件时,您必须链接由编程环境提供的专门的“导入库(import library)”。这些导入库包含了动态链接库名称和所有 Windows 函数调用的引用信息。链接程序使用该信息在.EXE 文件中构造一个表,当加载程序时,Windows 使用它将调用转换为 Windows 函数。

1.2 Windows 编程选项

为说明 Windows 编程的多种技术,本书提供了许多示例程序。这些程序使用 C 语言编写,并使用原始的 Windows API。我将这种方法称作“经典的”Windows 编程方法。这是我们在 1985 年为 Windows 1.0 编程的方法,它今天仍是编写 Windows 程序的有效方法。

1.2.1 API 和内存模式

对于程序员来说,操作系统是由本身的 API 定义的。API 包含了所有应用程序构造操作系统的函数调用,同时包含了相关的数据类型和结构。在 Windows 中,API 还意味着一个特殊的程序体系结构,我们将在每章的开头进行研究。

一般来说,Windows API 自 Windows 1.0 以来一直保持其一致性。具有 Windows 98 编程经验的 Windows 程序员会对 Windows 1.0 程序的源代码感觉非常熟悉。如果说 API 有什么变化的话,那就是它在不断地增强。Windows 1.0 支持不到 450 个函数调用,现在已有了几千种函数。

Windows API 和它的语法中的最大变化来自于从 16 位体系结构向 32 位体系结构转化的过程。Windows 从 1.0 到 3.1 使用 16 位的 Intel 8086、8088 和 286 微处理器的所谓分段内存模式,出于兼容性的考虑,从 386 开始的 32 位 Intel 微处理器也支持该模式。在这种模式下,微处理器寄存器的大小为 16 位,因此 C 的 int 数据类型也是 16 位宽。

在分段内存模式下，内存地址由两部分组成——一个 16 位段指针和一个 16 位偏移量指针。从程序员的角度看，这非常凌乱，并带来了 long 或 far 指针（包括段地址和偏移量地址）和 short 或 near 指针（包括带有假定段地址的偏移量地址）的区别。

从 Windows NT 和 Windows 95 开始，Windows 支持使用 Intel 386、486 和 Pentium 处理器的 32 位模式的 32 位平面内存模式。C 语言的 int 数据类型也扩展为 32 位的值。为 32 位版本 Windows 编写的程序使用在平面线性地址空间寻址的简单的 32 位指针值。

用于 16 位版本 Windows 的 API（Windows 1.0 到 Windows 3.1）现在被称作 Win16。用于 32 位版本 Windows 的 API（Windows 95、Windows 98 和所有版本的 Windows NT）现在被称作 Win32。许多函数调用在从 Win16 到 Win32 的转变中保持相同，但有些需要增强。例如，图像坐标点由 Win16 中的 16 位值变为 Win32 中的 32 位值。此外，某些 Win16 函数调用返回一个包含在 32 位整型值中的二维坐标点。这在 Win32 中不可能，因此增加的新函数调用以不同方式工作。

所有 32 位版本的 Windows 都支持 Win16 API（以确保和旧应用程序兼容）和 Win32 API（以运行新应用程序）。非常有趣的是，Windows NT 与 Windows 95 及 Windows 98 的工作方式不同。在 Windows NT 中，Win16 函数调用通过一个转换层被转化为 Win32 函数调用，然后被操作系统处理。在 Windows 95 和 Windows 98 中，该操作正相反：Win32 函数调用通过转换层转换为 Win16 函数调用，再由操作系统处理。

在同一时刻有两个不同的 Windows API 集（至少名称不同）。Win32s（"s"代表"subset（子集）"）是一个 API，允许程序员编写运行于 Windows 3.1 上的 32 位应用程序。该 API 仅支持已被 Win16 支持的 32 位函数版本。此外，Windows 95 API 一度被称作 Win32c（"c"代表"compatibility（兼容性）"），但该术语已被抛弃。

现在，Windows NT 和 Windows 98 都被认为能够支持 Win32 API。然而，每个操作系统依然都支持某些不被其他操作系统支持的功能特性。因为它们的相同之处是相当可观的，所以有可能编写在两个操作系统下都可运行的程序。而且，人们普遍认为这两个产品最终会合二为一。

1.2.2 语言选项

使用 C 语言和原始的 API 不是编写 Windows 98 程序的唯一方法。然而，这种方法却提供给您最佳的性能、最强大的功能，以及在发掘 Windows 特性方面最大的灵活性。可执行文件相对较小且运行时不要求外部库（自然，Windows DLL 自身除外）。最重要的是，不管您最终以什么方式编写 Windows 应用程序，熟悉 API 会使您对 Windows 内部有更深入的了解。

虽然我认为学习经典的 Windows 编程方法对任何 Windows 程序员都是重要的，但我没有必要建议使用 C 和 API 编写每个 Windows 应用程序。许多程序员，特别是那些

为公司内部编程或在家编写娱乐程序的程序员，喜欢轻松的开发环境，例如 Microsoft Visual Basic 或者 Borland Delphi(它是结合了面向对象技术的 Pascal 版本)。这些环境使程序员将精力集中于应用程序的用户界面和相关用户界面对象的代码上。要学习 Visual Basic，您也许需要参考微软出版社的一些其他图书，例如 Michael Halvorson 1996 年著的《Learn Visual Basic Now》。

在专业程序员中间——特别是那些编写商业应用程序的程序员——Microsoft Visual C++ 和 Microsoft 基础类库(MFC)是近年来流行的选择。MFC 在一个C++ 类集合中封装了许多 Windows 编程中的繁重内容。Jeff Prosise 的《Programming Windows with MFC, Second Edition》(微软出版社，1999 年)提供了 MFC 的指南。

最近，Internet 和 World Wide Web 的流行极大地推动了 Sun Microsystems 的 Java，这是一个受C++ 启发的与微处理器无关的编程语言，并且结合了可在几个操作系统平台上运行的用于编写图形应用程序的工具包。微软出版社有一本关于 Microsoft J++ (Microsoft 的 Java)开发工具的好书，《Programming Visual J++ 6.0》(1998 年)，由 Stephen R. Davis 著。

显然，很难说哪种方法更有利于编写 Windows 应用程序。更主要的是，也许是应用程序自身的特性决定了所使用的工具。不管您最后实际上使用什么工具编写代码，但学习 Windows API 将使您更深入地了解 Windows 工作的方式。Windows 是一个复杂的系统，在 API 上增加一个编程层并未减少它的复杂性，仅仅是掩盖了它，早晚您会碰到它。了解 API 会给您更好的补救机会。

在原始的 Windows API 之上的任何软件层都必定将您限制在全部功能的一个子集内。例如，您也许发现，使用 Visual Basic 编写应用程序非常理想，然而它不允许您做一个或两个基本任务。在这种情况下，您将不得不使用原始的 API 调用。API 定义了作为 Windows 程序员所需的一切。没有什么方法比直接使用 API 更万能的了。

MFC 是特别有疑问的。虽然它极大地简化了某些工作(例如 OLE)，我却经常发现当让它们按我所想的去工作时，会在其他特性(例如 Document/View 体系结构)上碰壁。MFC 还不是 Windows 编程所希望的灵丹妙药，很少有人认为它是一个好的面向对象设计的典型。MFC 程序员从他们使用的类定义如何工作中受益颇深，并会发现他们会经常参考 MFC 源代码，搞懂这些源代码是学习 Windows API 的好处之一。

1.2.3 编程环境

在本书中，假定您正使用 Microsoft Visual C++ 6.0，可以是标准版、专业版和企业版。经济的标准版足以应付本书中的编程。Visual C++ 也是 Visual Studio 6.0 中的一部分。

Microsoft Visual C++ 软件包中包括 C 编译器和其他编译及链接 Windows 程序所需的文件和工具等。它还包括 Visual C++ Developer Studio，一个可编辑源代码、交互建立

资源(如图标和对话框),以及编辑、编译、运行和调试程序的环境。

如果您正使用 Visual C++ 5.0,则需要为 Windows 98 和 Windows NT 5.0 更新头文件和导入库,这些东西可从 Microsoft 的 Web 站点上得到。在 http://www.microsoft.com/msdn/,选择 Downloads,然后选择 Platform SDK(软件开发工具包),您就能在选择的目录中下载和安装更新文件。要指引 Microsoft Developer Studio 访问这些目录,可以从 Tool 菜单项选择 Options,然后单击 Directories 标签。

Microsoft URL 上的 msdn 部分代表“Microsoft Developer Network(开发者网络)”。这是一个程序,向程序员提供了经常更新的 CD-ROM,其中包含了程序员在 Windows 开发中所需的最新东西。您也可以订阅 MSDN,这样就避免经常从 Microsoft 的 Web 站点下载。

1.2.4 API 文档

本书不是 Windows API 权威的正式文档的替代品。该文档不再以印刷形式出版,它只能从 CD-ROM 或 Internet 上得到。

在安装 Visual C++ 6.0 时,您将得到一个包括 API 文件的联机帮助系统。您可通过订阅 MSDN 或使用 Microsoft 的基于 Web 的联机帮助系统更新该文件。连接到http://www.microsoft.com/msdn/,然后选择 MSDN Library Online。

在 Visual C++ 6.0 中,从 Help 菜单项选择 Contents 项,调用 MSDN 窗口。API 文档按树形结构组织,查找标有 Platform SDK 的部分,所有在本书中引用的文档都来自于该部分。我将向您介绍如何从 Platform SDK 开始查找以斜杠分层嵌套的文件的位置。(我知道 Platform SDK 是整个 MSDN 知识中较为晦涩的部分,但我敢保证那是 Windows 编程的基本核心。)例如,对于如何在 Windows 程序中使用鼠标的文档,您可参考/Platform SDK/User Interface Services/User Input/Mouse Input。

我在前面提到 Windows 大致分为 Kernel、User 和 GDI 子系统。内核接口在/Platform SDK/Windows Base Services 中,用户接口函数在/Platform SDK/User Interface Services 中,GDI 位于/Platform SDK/Graphics and Multimedia Services/GDI 中。

1.3 编写第一个 Windows 程序

现在是编一些代码的时候了。为了便于对比,让我们以一个非常短的 Windows 程序和一个短的字符模式的程序开始。这会帮助我们找到使用开发环境并感受建立和编译程序机制的正确方向。

1.3.1 字符模式(Character-Mode)模型

程序员们喜爱的一本书是《The C Programming Language》(Prentice Hall,1978 年和 1988

年)，由 Brian W. Kernighan 和 Dennis M. Ritchie(亲切地称为 K&R)编著。该书的第一章以一个显示"hello,world"的 C 语言程序开始。

这里是在《The C Programming Language》第 1 版第 6 页中出现的程序：

```
main ( )
{
     printf ("hello, world \n") ;
}
```

以前 C 程序员在使用 printf 等 C 运行时库函数时，无需先声明它们。但这是 90 年代，我们愿意给编译器一个机会在我们的代码中标出错误。这里是在 K&R 第 2 版中修正的代码：

```
# include < stdio.h >

main ( )
{
     printf ("hello, world \n") ;
}
```

该程序仍然是那么短。它可通过编译并运行得很好，但当今许多程序员更愿意清楚地说明 main 函数的返回值，在这种情况下，ANSI C 规定该函数必须返回一个值：

```
# include < stdio.h >

int main ( )
{
     printf ("hello, world \n") ;

     return 0 ;
}
```

我们还可以包括 main 的参数，把程序搞得更长一些，但让我们姑且这样——包括一个 include 声明、程序的入口点、一个对运行时库函数的调用和一个 return 语句。

1.3.2 Windows 等价程序

Windows 关于"hello,world"程序的等价程序有和字符模式版本完全相同的组成。它有一个 include 声明、一个程序入口点、一个函数调用和一个返回语句。下面便是该程序：

```
/*------------------------------------------------------------
   HelloMsg.C —— Displays "Hello, Windows 98!" in a message box
                 (c) Charles Petzold, 1998
  ------------------------------------------------------------*/

# include < windows.h >
```

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     MessageBox (NULL, TEXT ("Hello, Windows 98!"), TEXT ("HelloMsg"), 0) ;

     return 0 ;
}
```

在剖析该程序之前，让我们看一下在 Visual C++ Developer Studio 中创建程序的机制。

首先，从 File 菜单中选择 New。在 New 对话框中，单击 Projects 标签，选择 Win32 Application。在 Location 域中，选择一个子目录，在 Project Name 域中，输入该项目的名称，此时该名称是 HelloMsg，这便是在 Location 域中显示的目录的子目录。应选中 Create New Workspace 单选钮，在 Platforms 部分应该指明 Win32，选择 OK。

将会出现一个标题为"Win32 Application-Step 1 Of 1"的对话框，指出要建立一个"Empty Project"，按下 Finish 按钮。

从 File 菜单中再次选择 New。在 New 对话框中，单击 Files 标签，选择C++ Source File。应选中 Add To Project 复选框，并应指明 HelloMsg。在 File Name 域中输入 HelloMsg.c，选择 OK。

现在您可输入上面所示的 HELLOMSG.C 文件，您也可以选择 Insert 菜单和 File As Text 选项，以从本书附带的 CD-ROM 上复制 HELLOMSG.C 的内容。

从结构上说，HELLOMSG.C 与 K&R 的"hello, world"程序是相同的。头文件STDIO.H 已被 WINDOWS.H 所代替，入口点 main 被 WinMain 所代替，而且 C 运行时库函数 printf 被 Windows API 函数 MessageBox 所代替。然而，在程序中有许多新东西，包括几个陌生的大写标识符。

让我们从头部开始。

1.3.3 头文件

HELLOMSG.C 以一个预处理指示开始，实际上在每个用 C 编写的 Windows 程序的头部都可看到：

```
#include <windows.h>
```

WINDOWS.H 是主要的包含文件，它包含了其他 Windows 头文件，这些头文件的某些也包含了其他头文件。这些头文件中最重要的和最基本的是：

- WINDEF.H 基本类型定义；
- WINNT.H 支持 Unicode 的类型定义；
- WINBASE.H 内核函数；

- WINUSER.H 用户接口函数;
- WINGDI.H 图形设备接口函数。

这些头文件定义了 Windows 的所有数据类型、函数调用、数据结构和常数标识符,它们是 Windows 文档中的一个重要部分。使用 Visual C++ Developer Studio 的 Edit 菜单中的 Find in Files 搜索这些头文件非常方便。您还可以在 Developer Studio 中打开这些头文件,并直接阅读它们。

1.3.4 程序入口点

正如在 C 程序中的入口点是函数 main 一样,Windows 程序的入口点是 WinMain,总是像这样出现:

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
```

上述入口点在/Platform SDK/User Interface Services/Windowing/Windows/Window Reference/Window Functions 中有说明。它在 WINBASE.H 中声明如下:

```
int
WINAPI
WinMain(
    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nShowCmd
    );
```

您会注意到我在 HELLOMSG.C 中做了许多小改动。第三个参数在 WINBASE.H 中定义为 LPSTR,我将它改为 PSTR。这两种数据类型都定义在 WINNT.H 中,作为指向字符串的指针。LP 前缀代表"长指针",这是 16 位 Windows 的一个创造。

我还在 WinMain 声明中改变了两个参数的名称。许多 Windows 程序中的变量名使用一种被称作"匈牙利表示法"的命名系统,该系统在变量名称前面增加了表示变量数据类型的短前缀,我将在第三章更详细地讨论这个概念。现在仅需记住:前缀 i 表示 int,sz 表示"以零结束的字符串"。

WinMain 函数声明为返回一个 int 值。WINAPI 标识符在 WINDEF.H 定义,语句如下:

```
#define WINAPI __stdcall
```

该语句指定了一个调用约定,包括如何生成机器代码以在堆栈中放置函数调用的参数。许多 Windows 函数调用声明为 WINAPI。

WinMain 的第一个参数被称作"实例句柄"。在 Windows 编程中,句柄仅是一个应用

程序用来识别某些事情的数字。在这种情况下,该句柄唯一地标识该程序,其他 Windows 函数需要用该句柄作为参数来调用该程序。在 Windows 的早期版本中,当同时运行同一程序多次时,您便创建了该程序的“多个实例”。同一应用程序的所有实例共享代码和只读的内存(通常是菜单和对话框模板之类的资源)。程序通过检查 hPrevInstance 参数就能够确定自身的其他实例是否正在运行。然后它可以越过某种繁杂的工作并从前面的实例将某些数据移到自己的数据区域。

在 32 位 Windows 版本中,该概念已被抛弃。WinMain 的第二个参数总是 NULL(定义为 0)。

WinMain 的第三个参数是用于运行程序的命令行。某些 Windows 应用程序使用这个参数以在程序启动时将文件加载到内存。WinMain 的第四个参数指出程序最初显示的方式,可以是正常地或者是最大化地充满整个窗口,或者是最小化显示在任务列表栏中。我们将在第三章中介绍使用该参数的方法。

1.3.5 MessageBox 函数

MessageBox 函数用于显示短信息。虽然,MessageBox 显示的小窗口不具有什么功能,实际上它被认为是一个对话框。

MessageBox 的第一个参数通常是窗口句柄,我们将在第三章介绍其含义。第二个参数是在消息框主体中显示的字符串,第三个参数是出现在消息框标题栏上的字符串。在 HELLMSG.C 中,这些文本字符串的每一个都被封装在一个“TEXT”宏中。通常您不必将所有字符串都封装在“TEXT”宏中,但如果想将您的程序转换为 Unicode 字符集,这确是一个好主意。我将在第二章详细讨论该问题。

MessageBox 的第四个参数可以是在 WINUSER.H 中定义的一组以前缀 MB_开始的常数的组合。您可从第一组中选择一个常数以指出希望在对话框中显示的按钮:

```
#define MB_OK                    0x00000000L
#define MB_OKCANCEL              0x00000001L
#define MB_ABORTRETRYIGNORE      0x00000002L
#define MB_YESNOCANCEL           0x00000003L
#define MB_YESNO                 0x00000004L
#define MB_RETRYCANCEL           0x00000005L
```

如果在 HELLOMSG 中将第四个参数设置为 0,则仅显示“OK”按钮。可以使用 C 语言的“或”(|)运算符将上面显示的一个常数与代表默认按钮的常数组合:

```
#define MB_DEFBUTTON1            0x00000000L
#define MB_DEFBUTTON2            0x00000100L
#define MB_DEFBUTTON3            0x00000200L
#define MB_DEFBUTTON4            0x00000300L
```

还可以使用一个常量指出消息框中图标的外观：

```
#define MB_ICONHAND              0x00000010L
#define MB_ICONQUESTION          0x00000020L
#define MB_ICONEXCLAMATION       0x00000030L
#define MB_ICONASTERISK          0x00000040L
```

其中某些图标有替代名称：

```
#define MB_ICONWARNING           MB_ICONEXCLAMATION
#define MB_ICONERROR             MB_ICONHAND
#define MB_ICONINFORMATION       MB_ICONASTERISK
#define MB_ICONSTOP              MB_ICONHAND
```

虽然只有少数其他 MB_ 常数，但您可以自己参考头文件或/Platform SDK/User Interface Services/Windowing/Dialog Boxes/Dialog Box Reference/Dialog Box Functions 里的文件。

在本程序中，MessageBox 返回数值 1，但更严格地说，它返回 IDOK，IDOK 在 WINUSER.H 中定义，等于 1。根据在消息框中显示的其他按钮，MessageBox 函数还可返回 IDYES、IDNO、IDCANCEL、IDABORT、IDRETRY 或 IDIGNORE。

这个小的 Windows 程序真地与 K&R 的"hello，world"程序等价吗？您也许认为不是，因为 MessageBox 函数并没有"hello，world"中 printf 函数所具有的潜在的格式化文本的能力。但我们将在下一章中看到编写类似 printf 的 MessageBox 版本的方法。

1.3.6 编译、链接和运行

当您准备好编译 HELLOMSG 时，可从 Build 菜单中选择 Build Hellomsg.exe，或者按 F7 键，或者在 Build 工具栏中选择 Build 图标。（该图标的外观显示在 Build 菜单中。如果当前没有显示 Build 工具栏，您可从 Tools 菜单中选择 Customize，然后选择 Toolbars 选项卡，选中 Build 或者 Build MiniBar。）

另一种方法，您可从 Build 菜单中选择 Execute Hellomsg.exe，或者按 Ctrl + F5，或者在 Build 工具栏上单击 Execute Program 图标（该图标看上去像一个红的感叹号），就会弹出一个消息框，询问是否编译该程序。

正常情况下，在编译阶段，编译器从 C 源代码文件生成一个.OBJ（目标）文件。在链接阶段，链接程序结合.OBJ 文件和.LIB（库）文件以建立.EXE（可执行）文件。通过在 Project 选项卡上选择 Settings，然后单击 Link 标签，可以查看这些库文件的列表。特别地，您会注意到 KERNEL32.LIB、USER32.LIB 和 GDI32.LIB。这些是三个主要 Windows 子系统的"导入库"。它们包含了动态链接库的名称以及绑定进.EXE 文件的引用信息。Windows 使用该信息处理程序对 KERNEL32.DLL、USER32.DLL 和 GDI32.DLL 动态链接库中函数的调用。

在 Visual C++ Developer Studio 中，您可用不同的配置编译和链接程序。默认情况下，它们是 Debug 和 Release。可执行文件被存放在以这些名称命名的子目录下。在 Debug 配置下，信息被附加到 .EXE 文件中，这些信息有助于调试程序和跟踪源代码。

如果您喜欢在命令行下工作，附带的 CD-ROM 包含所有示例程序的 .MAK(make)文件。（可通过 Tools 菜单选择 Options，再选择 Build 选项卡，来告诉 Developer Studio 生成 make 文件。这里有一个复选框需要选中。）您需要运行在 Developer Studio 的 BIN 子目录下的 VCVARS32.BAT 来设置环境变量。要从命令行执行 make 文件，可以转到 HELLOMSG 目录并执行：

```
NMAKE /f HelloMsg.mak CFG="HelloMsg - Win32 Debug"
```

或者

```
NMAKE /f HelloMsg.mak CFG="HelloMsg - Win32 Release"
```

然后您可通过输入：

```
DEBUG\HELLOMSG
```

或者

```
RELEASE\HELLOMSG
```

从命令行运行 .EXE 文件。

我已经在本书附带的 CD-ROM 中对工程文件中的默认 Debug 配置做了一个改动。在 Project Settings 对话框中，选择 C/C++ 选项卡后，在 Preprocessor Definitions 域中，我已定义了标识符 UNICODE。我将在下章中对此进行更多的解释。

第二章

Unicode 简介

在第一章中，我已经预告要讲述关于 C 语言的任何方面，您也许在传统字符模式编程中尚未遇到过它们，但它们却在 Microsoft Windows 编程中扮演着重要角色。宽字符集和 Unicode 差不多就是这样的问题。

简单地说，Unicode 是 ASCII 字符编码的一个扩展。在严格的 ASCII 中，每个字符用 7 位表示，计算机上普遍使用每字符 8 位；而 Unicode 使用全 16 位字符编码。这使得 Unicode 能够表示世界上所有的书写语言中可能用于计算机通讯的字符、象形文字和其他符号。Unicode 最初打算作为 ASCII 的补充，可能的话，最终将代替它。鉴于 ASCII 是计算机中最具支配地位的标准，所以这确是一个很高的目标。

Unicode 影响到了计算机工业的每个部分，也许对操作系统和编程语言的影响最大。从这方面来看，我们已经上路了。Windows NT 从底层支持 Unicode。(不幸的是，Windows 98 只是小部分支持 Unicode。)先天即被 ANSI 束缚的 C 编程语言通过对宽字符集的支持来支持 Unicode。下面将详细讨论这些内容。

自然，作为程序员，我们通常会面对许多繁重的工作。我已试图通过使本书中的所有程序“Unicode 化”去减轻其负担。其含义会随着本章对 Unicode 的讨论而清晰起来。

2.1 字符集简史

虽然不能确定人类开始讲话的时间，但书写已有大约 6000 年的历史了。实际上，早期书写的内容是象形文字。每个字符都对应于发声的字母表则出现于大约 3000 年前。虽然世界上使用的多种书写语言在某些时候都使用得相当好，但 19 世纪的几个发明者还是看到了更多的需求。Samuel F. B. Morse 在 1838 年到 1854 年间发明了电报，当时他还发明了一种电报上使用的代码。字母表中的每个字符对应于一系列短的和长的脉冲(点和长划)。虽然其中大小写字母之间没有区别，但数字和标点符号都有了自己的

代码。

Morse代码并不是以非手写的或印刷的象形文字来代表书写语言的第一个例子。1821年到1824年之间，年轻的Louis Braille受到在夜间读写信息的军用系统的启发，发明了一种代码，它用纸上突起的点作为代码来帮助盲人阅读。Braille代码实际上是一种6位代码，它把字符、常用字符组合、常用单词和标点进行编码。一个特殊的escape代码表示后续的字符代码应解释为大写。一个特殊的shift代码允许后续代码被解释为数字。

Telex代码，包括Baudot（以一个法国工程师命名，该工程师卒于1903年）以及一种被称为CCITT #2的代码(1931年被标准化)，都是包括字符转换和数字转换的5位代码。

2.1.1 美国标准

早期计算机的字符代码是从Hollerith卡片(不能折叠、卷曲或毁伤)发展而来的，该卡片由Herman Hollerith发明并首次在1890年的美国人口普查中使用。基于Hollerith代码的被称为BCDIC(Binary-Coded Decimal Interchange Code：二进制编码十进制交换编码)的6位字符代码在60年代逐步扩展为8位EBCDIC，并一直作为IBM大型机的标准，但其他地方并不使用。

美国信息交换标准码(ASCII：American Standard Code for Information Interchange)起始于50年代后期，并最终在1967年定案。开发ASCII的过程中，在代码位宽是6位、7位还是8位的问题上产生了很大的争议。从可靠性的观点来看，不应该使用转换字符，因此ASCII不能是6位编码，而且由于费用的原因也排除了8位版本的方案(当时每位的价格仍很昂贵)。这样，最终的代码就有26个小写字母、26个大写字母、10个数字、32个符号、33个控制代码和一个空格，总共128个代码。ASCII现在记录在ANSI X3.4-1986，代码字符集——用于信息交换的7位美国国家标准代码(7-Bit ASCII：7-Bit American National Standard Code for Information Interchange)，由美国国家标准协会(American National Standards Institute)发布。下页图2-1中所示的ASCII代码与ANSI文档中的格式非常相似。

您可以说出很多ASCII的优点。例如，26个字母代码是连续的(在EBCDIC代码中就不是这样的)；大写字母和小写字母可以通过改变一位数据而相互转化；10个数字的代码可以从数值本身方便地得到。(在BCDIC代码中，字符“0”的编码在字符“9”的后面！)

最棒的是，ASCII是一个非常可靠的标准。在键盘、视频显示卡、系统硬件、打印机、字体文件、操作系统，以及Internet上，其他标准都不如ASCII码这样流行，而且根深蒂固。

	0-	1-	2-	3-	4-	5-	6-	7-
-0	NUL	DLE	SP	0	@	P	`	p
-1	SOH	DC1	!	1	A	Q	a	q
-2	STX	DC2	"	2	B	R	b	r
-3	ETX	DC3	#	3	C	S	c	s
-4	EOT	DC4	$	4	D	T	d	t
-5	ENQ	NAK	%	5	E	U	e	u
-6	ACK	SYN	&	6	F	V	f	v
-7	BEL	ETB	'	7	G	W	g	w
-8	BS	CAN	(	8	H	X	h	x
-9	HT	EM	)	9	I	Y	i	y
-A	LF	SUB	*	:	J	Z	j	z
-B	VT	ESC	+	;	K	[	k	{
-C	FF	FS	,	<	L	\	l	|
-D	CR	GS	-	=	M	]	m	}
-E	SO	RS	.	>	N	^	n	~
-F	SI	US	/	?	O	_	o	DEL

图 2-1 ASCII 字符集

2.1.2 国际方面

ASCII 的最大问题就是缩写的首字母问题。ASCII 是一个真正的美国标准，所以它不能很好地满足其他讲英语国家的需要。例如英国的英镑符号（£）在哪里？

英语使用拉丁（或罗马）字母表。在使用拉丁字母表的书写语言中，英语中的单词通常很少需要重音符号（或读音符号）。即使那些有读音符号的英语单词也是符合传统惯例的，例如 coöperate 或者 résumé，拼写中没有读音符号也会被完全接受。

但美国南部、北部以及大西洋地区的许多国家在语言中使用读音符号很普遍。这些重音符号最初是为使拉丁字母表适合这些语言读音不同的需要。在远东或西欧的南部旅游，您会遇到根本不使用拉丁字母的语言，例如希腊语、希伯来语、阿拉伯语和俄语（使用斯拉夫字母表）。如果您向东走得更远，就会发现中国汉语采用的是象形字符，日本和朝鲜也采用这种字符。

ASCII 的历史开始于 1967 年，在此期间它主要致力于克服其自身限制来更适合于非美国英语的其他语言。例如，1967 年，国际标准化组织（ISO：International Standards Organization）推荐一个 ASCII 的变种，代码 0x40、0x5B、0x5C、0x5D、0x7B、0x7C 和 0x7D"为国家使用保留"，而代码 0x5E、0x60 和 0x7E 标为"当需要 8、9 或 10 位置作为国家代码时，可用于其他图形符号"。这显然不是一个最佳的国际解决方案，因为这并不能保证一致性。但这却显示了人们如何想尽办法为不同语言所用到的符号进行编码。

2.1.3 扩展 ASCII

在小型机开发的初期，就已经严格地建立了 8 位字节。因此，如果使用一个字节来保存字符，则可以由 128 个附加的字符来补充 ASCII。1981 年，当最初的 IBM PC 推出时，视频卡包括一个基于 ROM 的 256 个字符的字符集，其自身也就成为了 IBM 标准的一个重要组成部分。

最初的 IBM 扩展字符集包括某些带重音的字符和一个小写希腊字母表（在数学符号中非常有用），还包括一些块型和线型字符。附加的字符也被添加到 ASCII 控制字符的编码位置，这是因为控制字符的大多数不是必需的。

该 IBM 扩展字符集被烧进无数显示卡和打印机的 ROM 中，并被许多应用程序用于修饰其字符模式的显示方式。不过，该字符集并没有包括为所有使用拉丁字母表的西欧语言所需的足够多的带重音字符，而且也不适用于 Windows。Windows 不需要线型字符，因为它有一个完全图形化的系统。

在 Windows 1.0（1985 年 11 月发布）中，Microsoft 没有完全放弃 IBM 扩展字符集，但它已退居第二重要位置。因为基于 ANSI 草案和 ISO 标准，所以纯 Windows 字符集被称作“ANSI 字符集”。ANSI 草案和 ISO 标准最终成为 ANSI/ISO 8859-1-1987，即“American National Standard for Information Processing—8-Bit Single-Byte Coded Graphic Character Sets—Part 1：Latin Alphabet No 1”，通常也简写为“Latin 1”。

在 Windows 1.0 的《Programmer's Reference》中印出了 ANSI 字符集的最初版本，如图 2-2 所示。

	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0	□	□		0	@	P	`	p	□	□		°	À	Ð	à	ð
-1	□	□	!	1	A	Q	a	q	□	□	¡	±	Á	Ñ	á	ñ
-2	□	□	"	2	B	R	b	r	□	□	¢	²	Â	Ò	â	ò
-3	□	□	#	3	C	S	c	s	□	□	£	³	Ã	Ó	ã	ó
-4	□	□	$	4	D	T	d	t	□	□	¤	´	Ä	Ô	ä	ô
-5	□	□	%	5	E	U	e	u	□	□	¥	µ	Å	Õ	å	õ
-6	□	□	&	6	F	V	f	v	□	□	¦	¶	Æ	Ö	æ	ö
-7	□	□	'	7	G	W	g	w	□	□	§	·	Ç	□	ç	□
-8	□	□	(	8	H	X	h	x	□	□	¨	¸	È	Ø	è	ø
-9	□	□	)	9	I	Y	i	y	□	□	©	¹	É	Ù	é	ù
-A	□	□	*	:	J	Z	j	z	□	□	ª	º	Ê	Ú	ê	ú
-B	□	□	+	;	K	[	k	{	□	□	«	»	Ë	Û	ë	û
-C	□	□	,	<	L	\	l	|	□	□	¬	¼	Ì	Ü	ì	ü
-D	□	□	-	=	M	]	m	}	□	□	-	½	Í	Ý	í	ý
-E	□	□	.	>	N	^	n	~	□	□	®	¾	Î	Þ	î	þ
-F	□	□	/	?	O	_	o	DEL	□	□	¯	¿	Ï	ß	ï	ÿ

图 2-2 Windows ANSI 字符集（基于 ANSI/ISO 8859-1）

空心方框表示该位置未定义字符。这与 ANSI/ISO 8859-1 的最终定义一致。ANSI/ISO 8859-1 仅显示了图形字符,而没有控制字符,因此没有定义 DEL。此外,代码 0xA0 定义为一个非断开的空格(这意味着在编排格式时,该字符不用于折行),代码 0xAD 是一个软连字符(表示除非在行尾断开单词时使用,否则不显示)。此外,ANSI/ISO 8859-1 将代码 0xD7 定义为乘号(×),0xF7 为除号(÷)。Windows 中的某些字体也定义了从 0x80 到 0x9F 的某些字符,但这些不是 ANSI/ISO 8859-1 标准的一部分。

MS-DOS 3.3(1987 年 4 月发行)向 IBM PC 用户引进了代码页的概念,Windows 也使用此概念。代码页定义了字符的映射代码。最初的 IBM 字符集被称作代码页 437,或者"MS-DOS Latin US"。代码页 850 就是"MS-DOS Latin 1",它用附加的带重音字母(但不是图 2-2 所示的 Latin 1 ISO/ANSI 标准)代替一些线形字符。其他代码页被其他语言定义。最低的 128 个代码总是相同的;较高的 128 个代码取决于定义代码页的语言。

在 MS-DOS 中,如果用户为 PC 的键盘、显示卡和打印机指定了一个代码页,然后在 PC 上创建、编辑和打印文档,一切都很正常,每件事都会保持一致。然而,如果用户试图与使用不同代码页的用户交换文件,或者在机器上改变代码页,就会产生问题。字符代码与错误的字符相关联。应用程序能够将代码页信息与文档一起保存来试图减少问题的产生,但该策略包括了某些在代码页间转换的工作。

虽然代码页最初仅提供了不包括带重音符号字母的附加拉丁字符集,但最终代码页的较高的 128 个字符还是包括了完整的非拉丁字母,例如希伯来语、希腊语和斯拉夫语。自然,如此多样会导致代码页变得混乱;如果少数带重音的字母未正确显示,那么整个文字便会混乱不堪而不可阅读。

代码页的扩展正是基于所有这些原因,但是还很不够。斯拉夫语的 MS-DOS 代码页 855 与斯拉夫语的 Windows 代码页 1251 及斯拉夫语的 Macintosh 代码页 10007 不同。每个环境下的代码页都是为环境所做的标准字符集修正。IBM OS/2 也支持多种 EBCDIC 代码页。

但是且慢,更糟的事情还在后面。

2.1.4 双字节字符集

迄今为止,我们已经看到了 256 个字符的字符集。但中国、日本和朝鲜/韩国的象形文字符号有大约21 000个。如何容纳这些语言而仍保持和 ASCII 的某种兼容性呢?

解决方案(如果本用语正确的话)是双字节字符集(DBCS:double-byte character set)。DBCS 从 256 代码开始,就像 ASCII 一样。与任何表现良好的代码页一样,最初的 128 个代码是 ASCII。然而,较高的 128 个代码中的某些总是跟随着第二个字节。这两个字节一起(称作首字节和跟随字节)定义一个字符,通常是一个复杂的象形文字。

虽然中文、日文和朝鲜文共用一些相同的象形文字,但显然这三种语言是不同的,

而且经常是同一个象形文字在三种不同的语言中代表三件不同的事。Windows 支持 4 个不同的双字节字符集:代码页 932(日文)、936(简体中文)、949(朝鲜文)和 950(繁体中文)。只有为这些国家(地区)生产的 Windows 版本才支持 DBCS。

双字符集问题并不是说字符由两个字节代表。问题在于一些字符(特别是 ASCII 字符)由 1 个字节表示。这会引起附加的编程问题。例如,字符串中的字符数不能由字符串的字节数决定。必须剖析字符串来决定其长度,而且必须检查每个字节以确定它是否为双字节字符的首字节。如果有一个指向 DBCS 字符串中间的指针,那么该字符串前一个字符的地址是什么呢?常规的解决方案是从开始的指针分析该字符串!

2.1.5 Unicode 解决方案

我们面临的基本问题是世界上的书写语言不能简单地用 256 个 8 位代码表示。以前的解决方案包括代码页和 DBCS 已被证明是不能满足需要的,而且也是笨拙的。什么是真正的解决方案呢?

作为程序员,我们经历过这类问题。如果事情太多,用 8 位数值已经不能表示,那么我们就试更宽的值,例如 16 位值。这就是 Unicode 非常简单的原理。与混乱的 256 字符代码映射,以及含有一些单字节代码和一些双字节代码的双字节字符集不同,Unicode 是统一的 16 位系统,这样就允许表示65 536个字符。这对表示所有字符及世界上使用象形文字的语言,包括一系列的数学、符号和货币的集合来说是充裕的。

明白 Unicode 和 DBCS 之间的区别很重要。Unicode 据称是使用(特别在 C 编程语言环境中)"宽字符集"。Unicode 中的每个字符都是 16 位宽而不是 8 位宽。在 Unicode 中,8 位数值没有意义。相比之下,在双字节字符集中我们仍然处理 8 位数值。有些字节自身定义一个字符,而某些字节则需要和另一个字节共同定义一个字符。

尽管处理 DBCS 字符串非常杂乱,但是处理 Unicode 文本则更像处理有秩序的文本。您也许会高兴地知道前 128 个 Unicode 字符(16 位代码从 0x0000 到 0x007F)是 ASCII,而接下来的 128 个 Unicode 字符(代码从 0x0080 到 0x00FF)是 ISO 8859-1 对 ASCII 的扩展。Unicode 中不同部分的字符都同样基于现有的标准。这是为了便于转换。希腊字母表使用从 0x0370 到 0x03FF 的代码,斯拉夫语使用从 0x0400 到 0x04FF 的代码,美国使用从 0x0530 到 0x058F 的代码,希伯来语使用从 0x0590 到 0x05FF 的代码。中国、日本和韩国的象形文字(总称为 CJK)占用了从 0x3000 到 0x9FFF 的代码。

Unicode 的最大好处是这里只有一个字符集,没有一点含糊。Unicode 实际上是个人计算机行业中每个重要公司共同合作的结果,并且它与 ISO 10646-1 标准中的代码是一一对应的。Unicode 的重要参考文献是《The Unicode Standard, Version 2.0》(Addison-Wesley 出版社,1996 年)。这是一本非凡的书,它以其他文件少有的方式显示了世界上书写语言的丰富性和多样性。此外,该书还提供了 Unicode 开发的基本原理和细节。

Unicode 有缺点吗？当然有。Unicode 字符串占用的内存是 ASCII 字符串的两倍。(然而压缩文件有助于极大地减少文件所占的磁盘空间。)但也许最糟的缺点是：人们相对来说还不习惯使用 Unicode。作为程序员，我们有我们的工作。

2.2 宽字符和 C

对 C 程序员来说，16 位字符的想法的确让人扫兴。一个 char 和一个字节同宽是极少数可以确定的事情之一。少数程序员清楚 ANSI/ISO 9899-1990，"美国国家标准编程语言——C"(也称作"ANSI C")通过一个被称作"宽字符"的概念来支持用多个字节代表一个字符的字符集。这些宽字符与常用的字符完美地共存。

ANSI C 也支持多字节字符集，例如中文、日文和朝鲜文版本 Windows 支持的字符集。然而，这些多字节字符集按单字节值对待，其中一些字符改变了后续字符的含义。多字节字符集主要影响 C 运行时库函数。相比之下，宽字符比正常字符宽，而且会引起一些编译问题。

宽字符不需要 Unicode。Unicode 是一种可能的宽字符编码。然而，因为本书的焦点是 Windows 而不是 C 运行的理论，所以我将把宽字符和 Unicode 作为同义语。

2.2.1 char 数据类型

假定我们都非常熟悉在 C 程序中使用 char 数据类型来定义和存储字符和字符串。但为了便于理解 C 如何处理宽字符，让我们首先回顾一下可能在 Win32 程序中出现的标准字符定义。

下面的语句定义并初始化了一个只包含一个字符的变量：

```
char c = 'A' ;
```

变量 c 需要 1 个字节来保存，并将用十六进制数 0x41 初始化，这是字母 A 的 ASCII 代码。

您可以像这样定义一个指向字符串的指针：

```
char * p ;
```

因为 Windows 是一个 32 位操作系统，所以指针变量 p 需要用 4 个字节保存。您还可初始化一个指向字符串的指针：

```
char * p = "Hello!" ;
```

像前面一样，变量 p 也需要用 4 个字节保存。该字符串保存在静态内存中并占用 7 个字节——6 个字节保存字符串，另 1 个字节保存终止符 0。

您还可以像这样定义字符数组：

```
char a[10] ;
```

在这种情况下，编译器为该数组保留了 10 个字节的存储空间。表达式 sizeof (a) 将返回 10。如果该数值是全局的(即在所有函数外定义)，您可使用像下面的语句来初始化一个字符数组：

```
char a[] = "Hello!" ;
```

如果您将该数组定义为一个函数的局部变量，则必须将它定义为一个 static 变量，如下所示：

```
static char a[] = "Hello!" ;
```

无论哪种情况，字符串都存储在静态程序内存中，并在末尾添加 0，这样就需要 7 个字节的存储空间。

2.2.2 宽字符

Unicode 或者宽字符都没有改变 char 数据类型在 C 中的含义。char 继续表示 1 个字节的存储空间，sizeof (char) 继续返回 1。理论上，C 中 1 个字节可比 8 位长，但对我们大多数人来说，1 个字节(也就是 1 个 char)是 8 位宽。

C 中的宽字符基于 wchar_t 数据类型，它在几个头文件(包括 WCHAR.H)中都有定义，像这样：

```
typedef unsigned short wchar_t ;
```

因此，wchar_t 数据类型与无符号短整型相同，都是 16 位宽。

要定义包含一个宽字符的变量，可使用下面的语句：

```
wchar_t  c = 'A' ;
```

变量 c 是一个双字节值 0x0041，是 Unicode 表示的字母 A。(然而，因为 Intel 微处理器首先从最低位字节开始存储多字节数值，所以该字节实际上是以 0x41、0x00 的顺序保存在内存中。在检查 Unicode 文本的计算机存储时应注意这一点。)

您还可定义指向宽字符串的指针：

```
wchar_t * p = L"Hello!" ;
```

注意紧挨在第一个引号前面的大写字母 L(代表“long”)。这将告诉编译器该字符串按宽字符保存——即每个字符占用 2 个字节。通常，指针变量 p 要占用 4 个字节，而字符串变量需要 14 个字节——每个字符需要 2 个字节，末尾的 0 还需要 2 个字节。

同样，您还可以用下面的语句定义宽字符数组：

```
static wchar_t a[] = L"Hello!";
```

该字符串也需要 14 个字节的存储空间，sizeof (a)将返回 14。索引数组 a 可得到单独的字符。a[1] 的值是宽字符“e”，或者 0x0065。

虽然看上去更像一个印刷符号，但第一个引号前面的 L 非常重要，并且在两个符号之间必须没有空格。只有带有 L，编译器才知道您需要将字符串存为每个字符 2 字节。稍后，当我们看到在变量定义之外的其他地方使用宽字符串时，您还会遇到第一个引号前面的 L。幸运的是，如果忘记了包含 L，C 编译器通常会提出警告或错误信息。

您还可在单个字符前面使用 L 前缀来表示它们应被解释为宽字符。如下所示：

```
wchar_t c = L'A';
```

但通常这是不必要的。C 编译器无论如何会对该字符进行零扩充。

2.2.3 宽字符库函数

我们都知道如何获得字符串的长度。例如，如果我们已经像下面这样定义了一个字符串指针：

```
char * pc = "Hello!";
```

我们可以调用

```
iLength = strlen (pc);
```

这时变量 iLength 将等于 6，也就是字符串中的字符数。

太好了！现在让我们试着定义一个指向宽字符的指针：

```
wchar_t * pw = L"Hello!";
```

再次调用 strlen：

```
iLength = strlen (pw);
```

现在麻烦来了。首先，C 编译器会显示一条警告消息，可能是这样的内容：

```
'function' : incompatible types - from 'unsigned short *' to 'const char *'
```

这条消息的意思是：说明 strlen 函数被声明为接受 char 类型的指针，但它现在却接受了一个 unsigned short 类型的指针。您仍然可编译并运行该程序，但您会发现 iLength 等于 1。为什么？

字符串“Hello!”中的 6 个字符占用 16 位：

```
0x0048 0x0065 0x006C 0x006C 0x006F 0x0021
```

Intel 处理器在内存中将其存为：

```
48 00 65 00 6C 00 6C 00 6F 00 21 00
```

假定 strlen 函数正试图得到一个字符串的长度，并把第 1 个字节作为字符开始计数，但接着下一个字节是 0，则表示字符串结束。

这个小练习清楚地说明了 C 语言本身和运行时库函数之间的区别。编译器将字符串 L"Hello!" 解释为一组 16 位短整型数据，并将其保存在 wchar_t 数组中。编译器还处理数组索引和 sizeof 操作符，因此这些都能正常工作，但在链接时才添加运行时库函数，例如 strlen。这些函数认为字符串由单字节字符组成。遇到宽字符串时，函数就不像我们所希望那样运行了。

您可能要说："噢，太麻烦了！现在每个 C 库函数都必须重写以接受宽字符"。但事实上并不是每个 C 库函数都需要重写，只是那些有字符串参数的函数才需要重写，而且也不用由您来完成。这一切都已经做好了。

strlen 函数的宽字符版本是 wcslen(wide-character string length：宽字符串长度)，并且在 STRING.H(其中也声明了 strlen)和 WCHAR.H 中均有声明。strlen 函数声明如下：

```
size_t __cdecl strlen (const char * ) ;
```

而 wcslen 函数则声明如下：

```
size_t __cdecl wcslen (const wchar_t * ) ;
```

这时我们知道，要得到宽字符串的长度，可以调用

```
iLength = wcslen (pw) ;
```

该函数将返回字符串中的字符数 6。请记住，改成宽字节后，字符串的字符长度不改变，只是字节长度改变了。

您熟悉的所有带有字符串参数的 C 运行库函数都有宽字符版本。例如，wprintf 是 printf 的宽字符版本。这些函数在 WCHAR.H 和含有标准函数声明的头文件中声明。

2.2.4 维护单一源代码

当然，使用 Unicode 也有缺点。第一点也是最主要的一点是，程序中的每个字符串都将占用两倍的存储空间。此外，您将发现宽字符运行库中的函数比常规的函数大。出于这个原因，您也许想建立两个版本的程序——一个处理 ASCII 字符串，另一个处理 Unicode 字符串。最好的解决办法是维护既能按 ASCII 编译又能按 Unicode 编译的单一源代码文件。

虽然这只是一个小问题，但由于运行库函数有不同的名称，您也要用不同的方式定义字符，这将在处理前面有 L 的字符串文字时遇到麻烦。

一个办法是使用 Microsoft Visual C++ 包含的 TCHAR.H 头文件。该头文件不是

ANSI C 标准的一部分，因此那里定义的每个函数和宏定义的前面都有一条下划线。TCHAR.H 为需要字符串参数的标准运行库函数提供了一系列的替代名称(例如，_tprintf 和_tcslen)。有时这些名称也被称为"通用"函数名，因为它们既可以指向函数的 Unicode 版也可以指向非 Unicode 版。

如果定义了名为_UNICODE 的标识符，并且程序中包含了 TCHAR.H 头文件，那么，_tcslen 就定义为 wcslen：

```
#define _tcslen wcslen
```

如果没有定义 UNICODE，则 _tcslen 定义为 strlen：

```
#define _tcslen strlen
```

如此等等。TCHAR.H 还用一个新的数据类型 TCHAR 来解决两种字符数据类型的问题。如果定义了_UNICODE 标识符，那么 TCHAR 就是 wchar_t：

```
typedef wchar_t TCHAR ;
```

否则，TCHAR 就是 char：

```
typedef char TCHAR ;
```

现在开始讨论字符串文字中的 L 问题。如果定义了_UNICODE 标识符，那么一个称作__T 的宏就定义如下：

```
#define __T(x) L##x
```

这是相当晦涩的语法，但却属于 C 预处理的 ANSI C 标准。这一对 ## 称为"粘贴符号(token paste)"，它将字母 L 添加到宏参数上。因此，如果宏参数是"Hello!"，则 L##x 就是 L"Hello!"。

如果没有定义_UNICODE 标识符，则__T 宏只简单地定义如下：

```
#define __T(x) x
```

此外，还有两个宏与__T 定义相同：

```
#define _T(x)  __T(x)
#define _TEXT(x)  __T(x)
```

在 Win32 控制台程序中使用哪个宏，取决于您喜欢简洁还是详细。基本上，必须按下述方法在_T 或_TEXT 宏内定义字符串文字：

```
_TEXT ("Hello!")
```

这样做的话，如果定义了_UNICODE，那么该串将解释为宽字符的组合，否则解释为8位的字符。

2.3 宽字符和 Windows

Windows NT 从底层支持 Unicode。这意味着 Windows NT 内部使用由16位字符组成的字符串。因为世界上其他许多地方还不使用16位字符串，所以 Windows NT 必须经常将字符串在操作系统内进行转换。Windows NT 可运行 ASCII、Unicode 或者 ASCII 和 Unicode 混编的程序。即 Windows NT 支持不同的 API 函数调用，这些函数接受8位或16位的字符串。（我们将马上看到这是如何工作的。）

相对于 Windows NT，Windows 98 对 Unicode 的支持要少得多。只有很少的 Windows 98 函数调用支持宽字符串。（这些函数列在《Microsoft Knowledge Base article Q125671》中；它们包括 MessageBox。）如果要发行的程序中只有一个 .EXE 文件要求在 Windows NT 和 Windows 98 下都能运行，那么就不应该使用 Unicode，否则就不能在 Windows 98 下运行；程序尤其不能调用 Unicode 版的 Windows 函数。为了在将来发行 Unicode 版的程序时处于更有利的位置，您应试着编写既为 ASCII 又为 Unicode 编译的源代码。这就是本书中所有程序的编写方式。

2.3.1 Windows 头文件类型

正如您在第一章所看到的那样，Windows 程序都包括头文件 WINDOWS.H。该文件包括许多其他头文件，包括 WINDEF.H，该文件中有许多在 Windows 中使用的基本类型定义，而且它本身也包括 WINNT.H。WINNT.H 处理基本的 Unicode 支持。

WINNT.H 的前面包含 C 的头文件 CTYPE.H，这是 C 的众多头文件之一，其中包括 wchar_t 的定义。WINNT.H 定义了新的数据类型，称作 CHAR 和 WCHAR：

```
typedef char CHAR ;
typedef wchar_t WCHAR ;// wc
```

当需要定义8位字符或者16位字符时，建议在 Windows 程序中使用数据类型 CHAR 和 WCHAR。WCHAR 定义后面的注释是匈牙利标记法的提议：一个基于 WCHAR 数据类型的变量可在前面附加上字母 wc 以说明一个宽字符。

WINNT.H 头文件进而定义了可用作8位字符串指针的6种数据类型和4个可用作 const 8位字符串指针的数据类型。这里精选了头文件中一些实用的说明数据类型语句：

```
typedef CHAR * PCHAR, * LPCH, * PCH, * NPSTR, * LPSTR, * PSTR ;
typedef CONST CHAR * LPCCH, * PCCH, * LPCSTR, * PCSTR ;
```

前缀 N 和 L 表示"near"和"long"，指的是 16 位 Windows 中两种大小不同的指针。在 Win32 中 near 和 long 指针没有区别。

类似地，WINNT.H 定义了 6 种可作为 16 位字符串指针的数据类型和 4 种可作为 const 16 位字符串指针的数据类型：

```
typedef WCHAR * PWCHAR, * LPWCH, * PWCH, * NWPSTR, * LPWSTR, * PWSTR ;
typedef CONST WCHAR * LPCWCH, * PCWCH, * LPCWSTR, * PCWSTR ;
```

至此，我们有了数据类型 CHAR(一个 8 位的 char)和 WCHAR(一个 16 位的 wchar_t)，以及指向 CHAR 和 WCHAR 的指针。与 TCHAR.H 一样，WINNT.H 将 TCHAR 定义为一般的字符类型。如果定义了标识符 UNICODE(没有下划线)，则 TCHAR 和指向 TCHAR 的指针就分别定义为 WCHAR 和指向 WCHAR 的指针；如果没有定义标识符 UNICODE，则 TCHAR 和指向 TCHAR 的指针就分别定义为 char 和指向 char 的指针：

```
#ifdef UNICODE
typedef WCHAR TCHAR, * PTCHAR ;
typedef LPWSTR LPTCH, PTCH, PTSTR, LPTSTR ;
typedef LPCWSTR LPCTSTR ;
#else
typedef char TCHAR, * PTCHAR ;
typedef LPSTR LPTCH, PTCH, PTSTR, LPTSTR ;
typedef LPCSTR LPCTSTR ;
#endif
```

如果已经在某个头文件或者其他头文件中定义了 TCHAR 数据类型，那么 WINNT.H 和 WCHAR.H 头文件都能防止其重复定义。不过，无论何时在程序中使用其他头文件，都应在所有其他头文件之前包含 WINDOWS.H。

WINNT.H 头文件还定义了一个宏，该宏将 L 添加到字符串的第一个引号前。如果定义了 UNICODE 标识符，则一个称作__TEXT 的宏定义如下：

```
#define __TEXT(quote) L##quote
```

如果没有定义标识符 UNICODE，则像这样定义__TEXT 宏：

```
#define __TEXT(quote) quote
```

此外，TEXT 宏可这样定义：

```
#define TEXT(quote) __TEXT(quote)
```

这与 TCHAR.H 中定义_TEXT 宏的方法一样，只是不必理会下划线。我将在本书中使用这个宏的 TEXT 版本。

这些定义可使您在同一程序中混合使用 ASCII 和 Unicode 字符串,或者编写一个可被 ASCII 或 Unicode 编译的程序。如果您希望明确定义 8 位字符变量和字符串,请使用 CHAR、PCHAR(或者其他),以及带引号的字符串。为明确地使用 16 位字符变量和字符串,请使用 WCHAR、PWCHAR,并将 L 添加到引号前面。对于是 8 位还是 16 位取决于 UNICODE 标识符的定义的变量或字符串,要使用 TCHAR、PTCHAR 和 TEXT 宏。

2.3.2 Windows 函数调用

从 Windows 1.0 到 Windows 3.1 的 16 位 Windows 中,MessageBox 函数位于动态链接库 USER.EXE 中。在 Windows 3.1 软件开发包的 WINDOWS.H 中,MessageBox 函数定义如下:

```
int WINAPI MessageBox (HWND, LPCSTR, LPCSTR, UINT) ;
```

注意,函数的第二个、第三个参数是指向常数字符串的指针。当编译链接一个 Win16 程序时,Windows 并不处理 MessageBox 调用。程序.EXE 文件中的表,允许 Windows 将该程序的调用与 USER 库中的 MessageBox 函数动态链接起来。

32 位的 Windows(即所有版本的 Windows NT,以及 Windows 95 和 Windows 98)除了含有与 16 位兼容的 USER.EXE 以外,还含有一个称为 USER32.DLL 的动态链接库,该库含有 32 位用户接口函数的入口点,包括 32 位的 MessageBox。

但这是 Windows 支持 Unicode 的关键:在 USER32.DLL 中,没有 32 位 MessageBox 的函数的入口点。实际上,有两个入口点,一个名为 MessageBoxA(ASCII 版),另一个名为 MessageBoxW(宽字符版)。用字符串作参数的每个 Win32 函数都在操作系统中有两个入口点!幸运的是,您通常不必关心这个问题,程序中只需使用 MessageBox。与 TCHAR 头文件一样,每个 Windows 头文件都有我们需要的技巧。

下面是 MessageBoxA 在 WINUSER.H 中定义的方法。这与 MessageBox 早期的定义很相似:

```
WINUSERAPI int WINAPI MessageBoxA (HWND hWnd, LPCSTR lpText,
                                  LPCSTR lpCaption, UINT uType) ;
```

下面是 MessageBoxW:

```
WINUSERAPI int WINAPI MessageBoxW (HWND hWnd, LPCWSTR lpText,
                                  LPCWSTR lpCaption, UINT uType) ;
```

注意,MessageBoxW 函数的第二个和第三个参数是指向宽字符的指针。

如果需要同时使用并分别匹配 ASCII 和宽字符函数调用,那么您可在 Windows 程序中明确地使用 MessageBoxA 和 MessageBoxW 函数。但大多数程序员将继续使用 MessageBox。根据是否定义了 UNICODE,MessageBox 将与 MessageBoxA 或 MessageBoxW 一样。

在 WINUSER.H 中完成这一技巧时,代码相当琐碎:

```
# ifdef UNICODE
# define MessageBox MessageBoxW
# else
# define MessageBox MessageBoxA
# endif
```

这样,如果定义了 UNICODE 标识符,那么程序中所有的 MessageBox 函数调用实际上就是 MessageBoxW 函数;否则,就是 MessageBoxA 函数。

运行该程序时,Windows 将程序中不同的函数调用与不同的 Windows 动态链接库的入口点链接。虽然只有少数例外,但是在 Windows 98 中不能运行 Unicode 版的 Windows 函数。虽然这些函数有入口点,但通常返回错误代码。应用程序注意这些返回的错误并采取一些合理的动作。

2.3.3 Windows 的字符串函数

正如前面谈到的,Microsoft C 包括需要字符串参数的 C 运行时库函数的所有宽字符版本和普通版本。不过,Windows 复制了其中一部分。例如,下面是 Windows 定义的一组字符串函数,这些函数用来计算字符串长度、复制字符串、连接字符串和比较字符串:

```
iLength = lstrlen (pString) ;
pString = lstrcpy (pString1, pString2) ;
pString = lstrcpyn (pString1, pString2, iCount) ;
pString = lstrcat (pString1, pString2) ;
iComp = lstrcmp (pString1, pString2) ;
iComp = lstrcmpi (pString1, pString2) ;
```

这些函数与 C 库中对应的函数功能相同。如果定义了 UNICODE 标识符,那么这些函数将接受宽字符串,否则只接受常规字符串。宽字符串版本的 lstrlenW 函数可在 Windows 98 中运行。

2.3.4 在 Windows 中使用 printf

有字符模式、命令行 C 语言编程经历的程序员常常格外喜欢 printf 函数。即使有更简单的选择(例如 puts),但 printf 出现在 Kernighan 和 Ritchie 的"hello, world"程序中一点也不会令人惊奇。我们知道,增强"hello, world"最终需要 printf 的格式化输出,因此我们最好从头使用它。

但有个坏消息:在 Windows 程序中不能使用 printf。虽然 Windows 程序中可以使用大多数 C 的运行库——实际上,许多程序员更愿意使用 C 内存管理和文件 I/O 函数而

不是 Windows 中等价的函数——Windows 没有标准输入和标准输出的概念。在 Windows 程序中可使用 fprintf,但不能使用 printf。

还有一个好消息,那就是仍然可以使用 sprintf 及 sprintf 系列中的其他函数来显示文本。这些函数除了将内容格式化输出到的第一个参数所提供的字符串缓冲区以外,其功能与 printf 相同。然后便可对该字符串进行操作(例如将其传给 MessageBox)。

如果您从未使用过 sprintf (我第一次开始编写 Windows 程序时也没有使过此函数),这里有一个简短的追溯。printf 函数的声明如下:

```
int printf (const char * szFormat, ...) ;
```

第一个参数是一个格式字符串,后面是与格式字符串中的代码相对应的多个各种类型的参数。

sprintf 函数定义如下:

```
int sprintf (char * szBuffer, const char * szFormat, ...) ;
```

第一个参数是字符缓冲区;后面是一个格式字符串。Sprintf 不是将格式化结果写到标准输出,而是将其存入 szBuffer。该函数返回该字符串的长度。在字符模式编程中,

```
printf ("The sum of %i and %i is %i", 5, 3, 5+3) ;
```

的功能等价于

```
char szBuffer [100] ;
sprintf (szBuffer, "The sum of %i and %i is %i", 5, 3, 5+3) ;
puts (szBuffer) ;
```

在 Windows 中,使用 MessageBox 显示结果优于 puts。

几乎每个人都经历过,当格式字符串与被格式化的变量不匹配时,可能使 printf 运行错误并可能造成程序崩溃。使用 sprintf 时,您不但要担心这些,而且还有一个新的负担:您定义的字符串缓冲区必须足够大以存放结果。Microsoft 专用函数_snprintf 解决了这一问题,此函数引进了另一个参数,并以字符数表示出了缓冲区的大小。

vsprintf 是 sprintf 的一个变形,它只有三个参数。vsprintf 用于执行有多个参数的自定义函数,类似 printf 格式。vsprintf 的前两个参数与 sprintf 相同:一个用于保存结果的字符缓冲区和一个格式字符串。第三个参数是指向格式化参数数组的指针。实际上,该指针指向在堆栈中供函数调用的变量。va_list、va_start 和 va_end 宏(在 STDARG.H 中定义)帮助我们处理这个堆栈指针。本章最后的 SCRNSIZE 程序展示了使用这些宏的方法。使用 vsprintf 函数,sprintf 函数可以这样编写:

```
int sprintf (char * szBuffer, const char * szFormat, ...)
{
```

```
     int      iReturn ;
     va_list  pArgs ;

     va_start (pArgs, szFormat) ;
     iReturn = vsprintf (szBuffer, szFormat, pArgs) ;
     va_end (pArgs) ;

     return iReturn ;
}
```

va_start 宏将 pArg 设置为指向一个堆栈变量,而这个变量地址位于堆栈参数 szFormat 的上面。

由于许多早期的 Windows 程序使用了 sprintf 和 vsprintf,最终导致 Microsoft 向 Windows API 中增添了两个相似的函数。Windows 的 wsprintf 和 wvsprintf 函数在功能上与 sprintf 和 vsprintf 等价,但它们不能处理浮点格式。

当然,随着宽字符的引入,sprintf 函数在数量上得到了很大的发展,但这也产生了函数名称极为混乱的问题。表 2-1 列出了 Microsoft 的 C 运行库和 Windows 支持的所有 sprintf 函数。

表 2-1

	ASCII	宽字符	常规
参数的变量个数			
标准版	sprintf	swprintf	_stprintf
最大长度版	_snprintf	_snwprintf	_sntprintf
Windows 版	wsprintfA	wsprintfW	wsprintf
参数数组的指针			
标准版	vsprintf	vswprintf	_vstprintf
最大长度版	_vsnprintf	_vsnwprintf	_vsntprintf
Windows 版	wvsprintfA	wvsprintfW	wvsprintf

在宽字符版的 sprintf 函数中,字符串缓冲区被定义为宽字符串。在宽字符版的所有这些函数中,格式化字符串必须是一个宽字符串。但是,您必须确保所有传递给这些函数的其他字符串也必须由宽字符组成。

2.3.5 格式化消息框

程序 2-1 所示的 SCRNSIZE 程序演示了如何实现 MessageBoxPrintf 函数,该函数有许多参数,并能像 printf 那样编排它们的格式。

程序 2-1 SCRNSIZE 程序

SCRNSIZE.C

```
/*-----------------------------------------------------
   SCRNSIZE.C -- Displays screen size in a message box
                 (c) Charles Petzold, 1998
  -----------------------------------------------------*/

#include <windows.h>
#include <tchar.h>
#include <stdio.h>

int CDECL MessageBoxPrintf (TCHAR * szCaption, TCHAR * szFormat, ...)
{
     TCHAR   szBuffer [1024] ;
     va_list pArgList ;

          // The va_start macro (defined in STDARG.H) is usually equivalent to:
          // pArgList = (char *) &szFormat + sizeof (szFormat) ;

     va_start (pArgList, szFormat) ;

          // The last argument to wvsprintf points to the arguments

     _vsntprintf (szBuffer, sizeof (szBuffer) / sizeof (TCHAR),
                  szFormat, pArgList) ;

          // The va_end macro just zeroes out pArgList for no good reason

     va_end (pArgList) ;

     return MessageBox (NULL, szBuffer, szCaption, 0) ;
}
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     int cxScreen, cyScreen ;

     cxScreen = GetSystemMetrics (SM_CXSCREEN) ;
     cyScreen = GetSystemMetrics (SM_CYSCREEN) ;

     MessageBoxPrintf (TEXT ("ScrnSize"),
                       TEXT ("The screen is %i pixels wide by %i pixels high."),
                       cxScreen, cyScreen) ;
     return 0 ;
}
```

通过使用从 GetSystemMetrics 函数得到的信息,该程序以像素为单位显示了视频显示的宽度和高度。为获得 Windows 中不同对象的尺寸信息,GetSystemMetrics 是一个有用的函数。事实上,我将在第四章用 GetSystemMetrics 函数向您展示如何在一个 Windows 窗口中显示和滚动多行文本。

2.3.6 本书与国际化

为国际市场准备的 Windows 程序不仅仅要使用 Unicode。国际化超出了本书的范围,但在 Nadine Kano 所写的《Developing International Software for Windows 95 and Windows NT》(Microsoft 出版社,1995 年)一书中涉及了许多。

本书中的程序确保既可使用也可不使用定义的 UNICODE 标识符来编译。这包括对所有字符和字符串定义使用 TCHAR,对字符串文字使用 TEXT 宏,以及注意不要混淆字节和字符。例如,注意 SCRNSIZE 中的_vsntprintf 调用。第二个参数是以字符数表示的缓冲区大小。通常使用 sizeof (szBuffer)。但如果缓冲区中有宽字符,则返回的不是缓冲区的字符长度,而是缓冲区的字节大小。您必须用 sizeof(TCHAR)将其分开。

通常在 Visual C++ Developer Studio 中,可使用两种不同的配置来编译程序:Debug 和 Release。为简便起见,对本书的示例程序,我已修改了 Debug 配置,以便于定义 UNICODE 标识符。如果程序使用了需要字符串作参数的 C 运行时函数,那么_UNICODE 标识符也在 Debug 配置中定义。(要了解这是在哪里完成的,请从 Project 菜单中选择 Settings,然后单击 C/C++ 标签。)使用这种方式,这些程序就可以方便地被重新编译和链接,以供测试。

本书中的所有程序——无论是否为 Unicode 编译——都可在 Windows NT 下运行。除极少数情况外,本书中按 Unicode 编译的程序不能在 Windows 98 中运行,而非 Unicode 版可以在 Windows 98 中运行。本章和第一章的程序就是两个特例。MessageBoxW 是 Windows 98 支持的少数宽字符 Windows 函数之一。在 SCRNSIZE.C 中,如果用 Windows 函数 wprintf 代替了_vsntprintf(您还必须删除该函数的第二个参数),那么 SCRNSIZE.C 的 Unicode 版将不能在 Windows 98 下运行,这是因为 Windows 98 不执行 wprintfW。

在本书的后面(特别在第六章介绍键盘的使用时),我们将看到,编写能处理远东版 Windows 双字符集的 Windows 程序并非易事。本书没有说明如何去做,并且基于这个原因,本书中的某些非 Unicode 版本的程序在远东版的 Windows 下不能正常运行。这也是 Unicode 对将来的编程如此重要的一条理由。Unicode 允许程序更容易地跨越国界。

第三章

窗口和消息

在前两章,示例程序使用了函数 MessageBox 来向用户输出文本。MessageBox 函数创建一个"窗口"。在 Windows 中,"窗口"一词有确切的含义。一个窗口就是屏幕上的一个矩形区域,它接收用户的输入,并以文本或图形的格式显示输出内容。

MessageBox 函数创建一个窗口,但这只是一个功能有限的特殊窗口。消息窗口有一个带关闭按钮的标题栏、一个可选的图标、一行或多行文本,以及最多 4 个按钮。当然,必须选择 Windows 提供给您的图标与按钮。

MessageBox 函数固然非常有用,但我们将不对此做过多地讨论。我们不能在消息框中显示图形,而且也不能在消息框中添加菜单。要添加这些对象,就需要创建自己的窗口,现在就开始。

3.1 自己的窗口

创建窗口很简单,只需调用 CreateWindow 函数即可。

但实际上并非真得这么简单。虽然创建窗口的函数确实被命名为 CreateWindow,而且您也能在/Platform SDK/User Interface Services/Windowing/Windows/Window Reference/Window Functions 找到此函数的有关内容,但您将发现 CreateWindow 的第一个参数就是所谓的"窗口类名称",并且该窗口类连接所谓的"窗口过程"。在我们调用 CreateWindow 之前,有一点背景知识会对您大有帮助。

3.1.1 总体结构

进行 Windows 程序设计,实际上是在进行一种面向对象的程序设计(OOP)。这一点在 Windows 中使用得最多的对象身上表现最为明显。这种对象正是 Windows 之所以命名为"Windows"的原因,它具有人格化的特征,甚至可能会在您的梦中出现,这就是那个叫

做“窗口”的对象。

桌面上最明显的窗口就是应用程序窗口。这些窗口含有显示程序名称的标题栏、菜单，甚至可能还有工具栏和滚动条。另一类窗口是对话框，它可以有标题栏也可以没有标题栏。

装饰对话框表面的还有各式各样的按钮、单选钮、复选框、列表框、滚动条和文本输入区域。其中每一个小的可视对象都是一个窗口。更确切地说，这些都称为“子窗口”或“控件窗口”或“子窗口控件”。

作为对象，用户会在屏幕上看到这些窗口，并通过键盘和鼠标直接与它们进行交互操作。更有趣的是，程序员的观点与用户的观点极其类似。窗口以“消息”的形式接收窗口的输入，窗口也用消息与其他窗口通讯。对消息的理解将是学习如何编写 Windows 程序必须逾越的障碍之一。

这有一个 Windows 的消息示例：我们知道，大多数的 Windows 程序都有大小合适的应用程序窗口。也就是说，您能够通过鼠标拖动窗口的边框来改变窗口的大小。通常，程序将通过改变窗口中的内容来响应这种大小的变化。您可能会猜测（并且您也是正确的），是 Windows 本身而不是应用程序正在处理与用户重新调整窗口大小相关的全部代码。由于应用程序能改变其显示的格式，所以它也“知道”窗口大小改变了。

应用程序是如何知道用户改变了窗口的大小的呢？由于程序员习惯了常规的字符模式程序，操作系统没有设置将此类消息通知给用户的机制。问题的关键在于理解 Windows 所使用的体系结构。当用户改变窗口的大小时，Windows 给程序发送一条消息指出新窗口的大小。然后程序就可以调整窗口中的内容，以反映大小的变化。

“Windows 给程序发送消息。”我们希望读者不要对这句话视而不见。它到底表达了什么意思呢？我们在这里讨论的是程序代码，而不是一个电子邮件系统。操作系统怎么给程序发送消息呢？

其实，所谓“Windows 给程序发送消息”，是指 Windows 调用程序中的一个函数，该函数的参数描述了这个特定消息。这种位于 Windows 程序中的函数被称为“窗口过程”。

无疑，读者对程序调用操作系统的思路是很熟悉的。例如，程序在打开磁盘文件时就要使用有关的系统调用。读者所不习惯的，可能是操作系统调用程序，而这正是 Windows 面向对象体系结构的基础。

程序创建的每一个窗口都有相关的窗口过程。这个窗口过程是一个函数，既可以在程序中，也可以在动态链接库中。Windows 通过调用窗口过程来给窗口发送消息。窗口过程根据此消息进行处理，然后将控制返回给 Windows。

更确切地说，窗口通常是在“窗口类”的基础上创建的。窗口类标识了处理窗口消息的窗口过程。使用窗口类使多个窗口能够基于同一个窗口类，并且使用同一个窗口过程。例如，所有 Windows 程序中的所有按钮均基于同一个窗口类。这个窗口类与一个位于 Windows 动态链接库（处理所有的按钮窗口消息）的窗口过程相联系。

在面向对象的程序设计中,对象是代码与数据的组合。窗口是一种对象,其代码是窗口过程。数据是窗口过程保存的信息,以及 Windows 为每个窗口系统中那个窗口类保存的信息。

窗口过程处理给窗口发送的消息。这些消息经常是告知窗口,用户正使用键盘或鼠标进行输入。这正是按钮窗口知道它被“按下”的奥妙所在。在窗口大小改变或窗口表面需要重画时,由其他消息通知窗口。

Windows 程序开始执行后,Windows 为该程序创建一个“消息队列”。这个消息队列用来存放该程序可能创建的各种不同窗口的消息。程序中有一小段代码,叫做“消息循环”,用来从队列中取出消息,并且将它们发送给相应的窗口过程。有些消息直接发送给窗口过程,不用放入消息队列中。

如果您对这段有关 Windows 体系结构的过于简略的描述将信将疑,就让我们去看看在实际的程序中,窗口、窗口类、窗口过程、消息队列、消息循环和窗口消息是如何相互配合的。这或许对您会有些帮助。

3.1.2 HELLOWIN 程序

创建一个窗口首先需要注册一个窗口类,那需要一个窗口过程来处理窗口消息。这包括一段几乎出现在每一个 Windows 程序中的固定代码。程序 3-1 所示的 HELLOWIN 程序就是一个包括这段固定代码的简单程序。

程序 3-1 HELLOWIN 程序

HELLOWIN.C

```
/*------------------------------------------------------------
     HELLOWIN.C——Displays "Hello, Windows 98!" in client area
                 (c) Charles Petzold, 1998
  ------------------------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("HelloWin") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;
```

```
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }
     hwnd = CreateWindow (szAppName,                  // window class name
                          TEXT ("The Hello Program"), // window caption
                          WS_OVERLAPPEDWINDOW,        // window style
                          CW_USEDEFAULT,              // initial x position
                          CW_USEDEFAULT,              // initial y position
                          CW_USEDEFAULT,              // initial x size
                          CW_USEDEFAULT,              // initial y size
                          NULL,                       // parent window handle
                          NULL,                       // window menu handle
                          hInstance,                  // program instance handle
                          NULL) ;                     // creation parameters

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     HDC         hdc ;
     PAINTSTRUCT ps ;
     RECT        rect ;

     switch (message)
```

```
     }
     case WM_CREATE:
          PlaySound (TEXT ("hellowin.wav"), NULL, SND_FILENAME | SND_ASYNC) ;
          return 0 ;

     case WM_PAINT:
          hdc = BeginPaint (hwnd, &ps) ;

          GetClientRect (hwnd, &rect) ;

          DrawText (hdc, TEXT ("Hello, Windows 98!"), -1, &rect,
                    DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY:
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

该程序创建一个普通的应用程序窗口，如图 3-1 所示。在窗口客户区的中央显示“Hello, Windows 98!”。如果安装了声卡，那么您还可以听到相应的朗读声音。

图 3-1　HELLOWIN 窗口

请注意:如果使用 Microsoft Visual C++ 为此程序创建新项目,则您需要为连接程序做一个附加的对象库。从 Project 菜单选择 Setting 选项,然后选取 Link 选项卡。从 Category 列表框中选择 General,然后在 Object/Library Modules 文本框中添加 WINMM.LIB (Windows 多媒体)。这样做是因为 HELLOWIN 将使用多媒体功能调用,而默认的项目中又不包括多媒体对象库。否则连接程序报告错误信息,指出 PlaySound 函数不可用。

HELLOWIN 将访问文件 HELLOWIN.WAV,该文件在本书所附光盘的 HELLOWIN 文件夹中。执行 HELLOWIN.EXE 时,默认的文件夹必须是 HELLOWIN。在 Visual C++ 中执行此程序时,虽然将在 HELLOWIN 的 RELEASE 或 DEBUG 子文件夹中运行,但事情还是会这样的。

3.1.3 总体考虑

实际上,每一个 Windows 程序代码中都包括 HELLOWIN.C 程序的大部分。没人能真正记住编写此代码的全部语法;通常,Windows 程序员在开始一个新程序时总是复制一个现有的程序,然后再做相应的修改。您可以按此习惯免费使用本书附带光盘中的程序。

上面提到,HELLOWIN 将在其窗口的中央显示文本串。那不完全正确。文本实际显示在程序客户区的中央,它在图 3-1 中是标题栏和边界范围内的大片白色区域。这种特性对我们很重要;客户区就是程序自由绘图并且向用户交付可视输出的窗口区域。

如果您认真思考一下,将会发现虽然只有 80 行代码,这个窗口却令人惊讶地具有许多功能。您可以用鼠标按住标题栏,在屏幕上移动窗口;可以按住边框,改变窗口的大小。在窗口大小改变时,程序自动地将“Hello, Windows 98!”文本串重新定位在客户区域的中央。您可以按最大化按钮,放大 HELLOWIN 以充满整个屏幕;也可以按最小化按钮,将程序压缩为一个图标。您可以从系统菜单(标题栏最左端的小图标)中执行所有选项;也可以通过从系统菜单中选择 Close 选项,或者通过单击标题栏最右端的关闭按钮,或者通过双击标题栏最左端的图标来关闭窗口以终止程序的执行。

我们将在本章的余下部分对此程序作一个详细的回顾。当然,我们首先要从整体上看一下。

与前两章中的示例程序一样,HELLOWIN.C 也有一个 WinMain 函数,但它还有另外一个函数,名为 WndProc。这就是窗口过程。(在 Windows 程序员之间的交谈中,这被称为“win prock”。)注意,在 HELLOWIN.C 中没有调用 WndProc 的代码。然而,在 WinMain 中有对 WndProc 的声明,这就是函数要在程序顶部附近声明的原因。

Windows 函数调用

HELLOWIN 至少调用了 18 个 Windows 函数。下面以它们在 HELLOWIN 中出现的次序列出这些函数以及各自的简明描述:

- LoadIcon 加载图标供程序使用;
- LoadCursor 加载鼠标指针供程序使用;
- GetStockObject 获取一个图形对象,在这个例子中,是获取绘制窗口背景的刷子;
- RegisterClass 为程序窗口注册窗口类;
- MessageBox 显示消息框;
- CreateWindow 根据窗口类创建一个窗口;
- ShowWindow 在屏幕上显示窗口;
- UpdateWindow 指示窗口刷新自身;
- GetMessage 从消息队列中获取消息;
- TranslateMessage 转换某些键盘消息;
- DispatchMessage 将消息发送给窗口过程;
- PlaySound 播放一个声音文件;
- BeginPaint 开始窗口绘制;
- GetClientRect 获取窗口客户区的尺寸;
- DrawText 显示文本串;
- EndPaint 结束窗口绘制;
- PostQuitMessage 在消息队列中插入一条"退出"消息;
- DefWindowProc 执行默认的消息处理。

这些函数均在平台 SDK 文档中说明,并在不同的头文件中声明,其中绝大多数在 WINUSER.H 中声明。

大写的标识符

读者可能注意到,在 HELLOWIN.C 中有很多大写的标识符,这些标识符都是在 Windows 的头文件中定义的。其中有些标识符包含由两个字母或者三个字母组成并后跟一个下划线的前缀:

CS_HREDRAW	DT_VCENTER	SND_FILENAME
CS_VREDRAW	IDC_ARROW	WM_CREATE
CW_USEDEFAULT	IDI_APPLICATION	WM_DESTROY
DT_CENTER	MB_ICONERROR	WM_PAINT
DT_SINGLELINE	SND_ASYNC	WS_OVERLAPPEDWINDOW

这些都是简单的数值常量。前缀指示该常量所属的类别,如下页表 3-1 所示。

在编写 Windows 程序时,不用去记这些数值常量。实际上,Windows 中使用的每个数值常量在头文件中均有相应的标识符定义。

表 3-1

前缀	类别
CS	类风格选项
CW	创建窗口选项
DT	绘制文本选项
IDI	图标 ID 号
IDC	光标 ID 号
MB	消息框选项
SND	声音选项
WM	窗口消息
WS	窗口风格

新的数据类型

HELLOWIN.C 中的其他标识符是新的数据类型,也在 Windows 头文件中使用 typedef 语句或者 #define 语句加以定义。这最初是为了便于将 Windows 程序从原来的 16 位系统上移植到未来的基于 32 位(或者其他)技术的操作系统上。这种作法并不如当时每个人想象的那样顺利,但是这种概念基本上是正确的。

有时这些新的数据类型只是为了方便起见的一个缩写。例如,用于 WndProc 的第二个参数的 UINT 数据类型只是一个 unsigned int (无符号整数),在 Windows 98 中,这是一个 32 位的值。用于 WinMain 的第三个参数的 PSTR 数据类型是指向一个字符串的指针,即是一个 char * 。

其他数据类型的含义不太明显。例如,WndProc 的第三和第四个参数分别被定义为 WPARAM 和 LPARAM,这些名字的来源有点历史背景:当 Windows 是一种 16 位系统时,WndProc 的第三个参数被定义为一个 WORD,这是一个 16 位的无符号短整数,而第四个参数被定义为一个 LONG,这是一个 32 位有符号的长整数,从而导致了对单词"PARAM"加前缀"W"和"L"。当然,在 32 位的 Windows 中,WPARAM 被定义为一个 UINT,而 LPARAM 被定义为一个 LONG(就是 C 中的 long 数据类型),因此窗口过程的这两个参数都是 32 位的值。这也许有点奇怪,因为 WORD 数据类型在 Windows 98 中仍然被定义为 16 位的无符号整数,因此"PARAM"的"W"在某种意义上是不正确的名字。

WndProc 函数返回一个类型为 LRESULT 的值,该值简单地被定义为一个 LONG。WinMain 函数被指定为一个 WINAPI 类型(在头文件中定义的所有 Windows 函数都被指定这种类型),而 WndProc 函数被指定为一个 CALLBACK 类型。这两个标识符都被定义为__stdcall,指在 Windows 本身和用户的应用程序之间发生的函数调用的特殊调用序列。

HELLOWIN 还使用了 Windows 头文件中定义的 4 种数据结构(我们将在本章稍后加以讨论)。这些数据结构如表 3-2 所示。表中前面两个数据结构在 WinMain 中使用,

分别定义了两个名为 msg 和 wndclass 的结构,后面两个数据结构在 WndProc 中使用,分别定义了 ps 和 rect 结构。

表 3-2

结构	含义
MSG	消息结构
WNDCLASS	窗口类结构
PAINTSTRUCT	绘图结构
RECT	矩形结构

句柄简介

最后,还有 3 个大写标识符(见表 3-3),用于不同类型的"句柄"。

表 3-3

标识符	含义
HINSTANCE	实例(程序自身)句柄
HWND	窗口句柄
HDC	设备描述表句柄

句柄在 Windows 中使用非常频繁。在本章结束之前,我们将遇到 HICON(图标句柄)、HCURSOR(鼠标指针句柄)和 HBRUSH(图形刷句柄)。

句柄是一个数(通常为 32 位),它代表一个对象。Windows 中的句柄类似传统 C 或者 MS-DOS 程序设计中使用的文件句柄。程序几乎总是通过调用 Windows 函数获取句柄。程序在其他 Windows 函数中使用这个句柄,以引用它代表的对象。句柄的实际值对程序来说是无关紧要的。但是,向您的程序提供句柄的 Windows 模块知道如何使用它来引用相应的对象。

匈牙利表示法

读者可能注意到,HELLOWIN.C 中有一些变量的名字显得很古怪。如 szCmdLine,它作为参数传递给 WinMain。

许多 Windows 程序员使用一种叫做"匈牙利表示法"的变量命名约定。这是为了纪念传奇性的 Microsoft 程序员 Charles Simonyi。非常简单,变量名以一个或者多个小写字母开始,这些字母表示变量的数据类型。例如,szCmdLine 中的 sz 代表"以 0 结尾的字符串"。在 hInstance 和 hPrevInstance 中的 h 前缀表示"句柄";在 iCmdShow 中的 i 前缀表示"整数"。WndProc 的后两个参数也使用匈牙利表示法。正如我在前面已经解释过的,尽管 wParam 应该更适当地被命名为 uiParam(代表"无符号整数"),但是因为这两个参数

是使用数据类型 WPARAM 和 LPARAM 定义的,因此保留了它们传统的名字。

在命名结构变量时,可以用结构名(或者结构名的一种缩写)的小写作为变量名的前缀,或者用作整个变量名。例如,在 HELLOWIN.C 的 WinMain 函数中,msg 变量是 MSG 类型的结构;wndclass 是 WNDCLASSEX 类型的一个结构。在 WndPmc 函数中,ps 是一个 PAINTSTRUCT 结构,rect 是一个 RECT 结构。

匈牙利表示法能够帮助程序员及早发现并避免代码中的错误。由于变量名既描述了变量的作用,又描述了其数据类型,这就比较容易避免产生数据类型不匹配的错误。

表 3-4 列出了在本书中将经常使用的变量前缀。

表 3-4

前缀	数据类型
c	char 或 WCHAR 或 TCHAR
by	BYTE (无符号字符)
n	short
i	int
x, y	int 分别用作 x 坐标和 y 坐标
cx, cy	int 分别用作 x 长度和 y 长度;C 代表"count"
b 或 f	BOOL (int);f 代表"flag"
w	WORD (无符号 short)
l	LONG (长整数)
dw	DWORD (无符号长整数)
fn	function(函数)
s	string(串)
sz	以 0 字节结尾的串
h	句柄
p	指针

3.1.4 注册窗口类

窗口总是在窗口类的基础上创建的,窗口类用以标识处理窗口消息的窗口过程。

可以在单个窗口类的基础上创建多个窗口。例如,Windows 中的所有按钮窗口——包括下压按钮、复选框,以及单选按钮——都是基于同一窗口类创建的。窗口类定义了窗口过程和基于此类创建的窗口的其他一些特征。在创建窗口时,要定义一些窗口所独有的特征。

在为程序创建窗口之前,必须首先调用 RegisterClass 注册一个窗口类。该函数只需要一个参数,即一个指向类型为 WNDCLASS 的结构指针。此结构包括两个指向字符串的域,因此结构在 WINUSER.H 头文件中定义了两种不同的方式,第一个是 ASCII 版的 WNDCLASSA:

```
typedef struct tagWNDCLASSA
{
     UINT          style ;
     WNDPROC       lpfnWndProc ;
     int           cbClsExtra ;
     int           cbWndExtra ;
     HINSTANCE     hInstance ;
     HICON         hIcon ;
     HCURSOR       hCursor ;
     HBRUSH        hbrBackground ;
     LPCSTR        lpszMenuName ;
     LPCSTR        lpszClassName ;
}
WNDCLASSA, * PWNDCLASSA, NEAR * NPWNDCLASSA, FAR * LPWNDCLASSA ;
```

在这里提示一下数据类型和匈牙利表示法:其中的 lpfn 前缀代表"指向函数的长指针"。(我们可以回忆出在 Win32 API 中,长指针和短(或者近)指针是没有区别的。这只是 16 位 Windows 的遗物。)cb 前缀代表"count of bytes(字节数)",常用于表示字节长度的变量。h 前缀是一个句柄,而 hbr 前缀代表"一个刷子的句柄"。lpsz 前缀代表"指向以 0 结尾的字符串的长指针"。

Unicode 版的结构定义如下:

```
typedef struct tagWNDCLASSW
{
     UINT          style ;
     WNDPROC       lpfnWndProc ;
     int           cbClsExtra ;
     int           cbWndExtra ;
     HINSTANCE     hInstance ;
     HICON         hIcon ;
     HCURSOR       hCursor ;
     HBRUSH        hbrBackground ;
     LPCWSTR       lpszMenuName ;
     LPCWSTR       lpszClassName ;
}
WNDCLASSW, * PWNDCLASSW, NEAR * NPWNDCLASSW, FAR * LPWNDCLASSW ;
```

与前者唯一的区别在于最后两个域定义为指向宽字符串常量,而不是指向 ASCII 字符串常量。

WINUSER.H 定义了 WNDCLASSA 和 WNDCLASSW 结构(以及指向结构的指针)以后,头文件基于对 UNICODE 标识的解释,定义了 WNDCLASS 和指向 WNDCLASS 的指针(包括一些向后兼容的代码):

```
#ifdef UNICODE
typedef WNDCLASSW WNDCLASS ;
typedef PWNDCLASSW PWNDCLASS ;
```

```
typedef NPWNDCLASSW NPWNDCLASS ;
typedef LPWNDCLASSW LPWNDCLASS ;
#else
typedef WNDCLASSA WNDCLASS ;
typedef PWNDCLASSA PWNDCLASS ;
typedef NPWNDCLASSA NPWNDCLASS ;
typedef LPWNDCLASSA LPWNDCLASS ;
#endif
```

本书后面列出结构时，将只列出在功能上等价的结构定义，对于 WNDCLASS 则如：

```
typedef struct
{
     UINT          style ;
     WNDPROC       lpfnWndProc ;
     int           cbClsExtra ;
     int           cbWndExtra ;
     HINSTANCE     hInstance ;
     HICON         hIcon ;
     HCURSOR       hCursor ;
     HBRUSH        hbrBackground ;
     LPCTSTR       lpszMenuName ;
     LPCTSTR       lpszClassName ;
}
WNDCLASS, * PWNDCLASS ;
```

我也不再着重说明对各个指针的定义。没有理由让那些以 LP 和 NP 为前缀的不同类型来搅乱您的程序代码。

在 WinMain 中为 WNDCLASS 定义一个结构，通常像这样：

```
WNDCLASS wndclass ;
```

然后，可以初始化该结构的 10 个域，并调用 RegisterClass。

在 WNDCLASS 结构中最重要的两个域是第二个和最后一个，第二个域（lpfnWndProc）是所有基于这个类来创建的窗口所使用的窗口过程的地址。在 HELLOWIN.C 中，这个过程是 WndProc 函数。最后一个域是窗口类的文本名。程序员可以随意定义其名称。在只创建一个窗口的程序中，窗口类名称通常设置为程序名称。

其他域按下述的方法描述了窗口类的一些特征。让我们依次看一下 WNDCLASS 结构中的每个域。

语句

```
wndclass.style = CS_HREDRAW | CS_VREDRAW ;
```

使用 C 的按位"或"操作符组合了两个"类风格"标识符。在头文件 WINUSER.H 中，已定义了一个以 CS 为前缀的标识符集合：

```
#define CS_VREDRAW            0x0001
#define CS_HREDRAW            0x0002
#define CS_KEYCVTWINDOW       0x0004
#define CS_DBLCLKS            0x0008
#define CS_OWNDC              0x0020
#define CS_CLASSDC            0x0040
#define CS_PARENTDC           0x0080
#define CS_NOKEYCVT           0x0100
#define CS_NOCLOSE            0x0200
#define CS_SAVEBITS           0x0800
#define CS_BYTEALIGNCLIENT    0x1000
#define CS_BYTEALIGNWINDOW    0x2000
#define CS_GLOBALCLASS        0x4000
#define CS_IME                0x00010000
```

由于每个标识符都在复合值中设置了一位,所以按这种方式定义的标识符通常称为“位标志”。通常只使用少数的类风格。HELLOWIN 中用到的这两个标识符表明,所有基于此类而创建的窗口,每当窗口的水平方向尺寸(CS_HREDRAW)或者垂直方向尺寸(CS_VREDRAW)改变之后,要完全刷新窗口。改变 HELLOWIN 的窗口大小,可以看到文本串仍然显示在窗口的中央。这两个标识符确保了这一点。不久我们就将看到窗口过程是如何被告知这种窗口大小的变化的。

WNDCLASS 结构的第二个域由以下语句进行初始化:

```
wndclass.lpfnWndProc = WndProc ;
```

这条语句将这个窗口类的窗口过程设置为 WndProc,即 HELLOWIN.C 中的第二个函数。这个过程将处理基于这个窗口类创建的所有窗口的全部消息。在 C 语言中,像这样在语句中使用函数名时,实际引用提供的是指向函数的指针。

下面两个域用于在类结构和 Windows 内部保存的窗口结构中预留一些额外空间:

```
wndclass.cbClsExtra = 0 ;
wndclass.cbWndExtra = 0 ;
```

程序可以根据需要来使用预留的空间。HELLOWIN 没有使用它们,所以赋值为 0。否则,和匈牙利表示法所指示的一样,这个域将被置为“字节数”。(在第七章的程序 CHECKER3 将使用 cbWndExtra 域。)

下一个域就是程序的实例句柄(它也是 WinMain 的参数之一):

```
wndclass.hInstance = hInstance ;
```

语句

```
wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION) ;
```

为所有基于这个窗口类建立的窗口设置一个图标。图标是一个小的位图图像，它对于用户代表程序，将出现在 Windows 任务栏中和窗口的标题栏的左端。在本书的后面，您将学习如何为您的 Windows 程序定制图标。现在，为了方便起见，我们将使用预先定义的图标。

要获取预先定义的图标句柄，可以将第一个参数设置为 NULL 来调用 LoadIcon。在加载程序员定制的图标时(图标应该保存磁盘上的 .EXE 程序文件中)，这个参数应该被设置为程序的实例句柄 hInstance。第二个参数标识图标。对于预先定义的图标，此参数是以 IDI 开始的标识符("ID 代表图标")，标识符在 WINUSER.H 中定义。IDI_APPLICATION 图标是一个简单的窗口小图形。LoadIcon 函数返回该图标的句柄。我们并不关心这个句柄的实际值，它只用于设置 hIcon 域的值。该域在 WNDCLASS 结构中定义为 HICON 类型，此类型名的含义为"handle to an icon(图标句柄)"。

语句

```
wndclass.hCursor = LoadCursor (NULL, IDC_ARROW) ;
```

与前一条语句非常相似。LoadCursor 函数加载一个预先定义的鼠标光标(命名为 IDC_ARROW)，并返回该光标的句柄。该句柄被赋给 WNDCLASS 结构的 hCursor 域。当鼠标光标在基于这个类创建的窗口的客户区域上出现时，它变成一个小箭头。

下一个域指定基于这个类创建的窗口背景颜色。hbrBackground 域名中的 hbr 前缀代表"handle to a brush(刷子句柄)"。刷子是图形学上的一个术语，指用来填充一个区域的着色像素模式。Windows 有几个标准刷子，也称为"备用(stock)"刷子。这里所示的 GetStockObject 调用将返回一个白色刷子的句柄：

```
wndclass.hbrBackground = GetStockObject (WHITE_BRUSH) ;
```

这意味着窗口客户区域的背景完全为白色，这是一种极其普遍的做法。

下一个域指定窗口类菜单。HELLOWIN 没有应用程序菜单，所以该字段被设置为 NULL：

```
wndclass.lpszMenuName = NULL ;
```

最后，必须给出一个类名。对于小程序，类名可以与程序名相同，即存放在 szAppName 变量中的"HelloWin"串。

```
wndclass.lpszClassName = szAppName ;
```

该串由 ASCII 字符组成还是由 Unicode 字符组成取决于是否定义了 UNICODE 标识符。

在初始化结构的 10 个域后，HELLOWIN 调用 RegisterClass 来注册这个窗口类。该函数只有一个参数，即指向 WNDCLASS 结构的指针。实际上，RegisterClassA 函数将获得一个指向 WNDCLASSA 结构的指针，而 RegisterClassW 函数将获得一个指向 WND-

CLASSW 结构的指针。程序要使用哪个函数来注册窗口类，取决于发送给窗口的消息包含 ASCII 文本还是 Unicode 文本。

现在有一个问题：如果用定义的 UNICODE 标识符编译了程序，程序将调用 RegisterClassW。该程序可以在 Microsoft Windows NT 中很好地运行。但如果此程序运行在 Windows 98 上，则运行 RegisterClassW 函数并未真地投入运行。函数有一个入口点，但它通过函数调用后返回 0 表明有错误发生。对于运行在 Windows 98 下的 Unicode 来说，这是一个通知用户有问题并终止运行的好机会。这是本书中多数程序处理 RegisterClass 函数调用的方法：

```
if (!RegisterClass (&wndclass))
{
     MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                 szAppName, MB_ICONERROR) ;
     return 0 ;
}
```

由于 MessageBoxW 是可在 Windows 98 环境下运行的几个 Unicode 函数之一，所以其运行正常。

当然，这代码段假定由于某种原因，RegisterClass 不是默认的，就像 WNDCLASS 结构中 lpfnWndProc 字段的 NULL 一样。GetLastError 函数会帮助您确定在这样的情况下产生错误的原因。GetLastError 是 Windows 中常用的函数，它可用于当函数调用失败时获得扩展的错误信息。不同函数的文档将指出您是否能够用 GetLastError 来获得此信息。在 Windows 98 中调用 RegisterClassW 时，GetLastError 将返回 120。在 WINERROR.H 中您可以看到，值 120 与标识符 ERROR_CALL_NOT_IMPLEMENTED 相等。您也可以在 /Platform SDK/Windows Base Services/Debugging and Error Handling/Error Codes/System Errors-Numerical Order 查看错误。

一些 Windows 程序员喜欢在查错时检查每一次函数调用的返回值。这的确有一定的意义。相信您非常熟悉您自己的常规方法：通常在分配内存之后检查错误。而许多 Windows 函数需要分配内存。例如，RegisterClass 需要分配内存，以保存关于窗口类的信息。这样，无论如何也要检查函数。另外，如果由于 RegisterClass 不能分配需要的内存，它也将失败。Windows 也可能要异常终止。

在本书的示例程序中，我做了最低的错误检查。这不是由于我认为错误检查不是一个好方法，而是因为这会导致我们从程序举例分心。

最后，一条历史性的经验是：在一些 Windows 示例程序中，您可能在 WinMain 中看到以下代码：

```
if (!hPrevInstance)
{
     wndclass.cbStyle = CS_HREDRAW | CS_VREDRAW ;
```

```
        [other wndclass initialization]
        RegisterClass (&wndclass) ;
}
```

这是出于“旧习难改”的原因。在16位的Windows中,如果您启动正在运行的程序的一个新实例,WinMain的hPrevInstance参数将是前一个实例的实例句柄。为节省内存,两个或多个实例就允许共享相同的窗口类。这样,窗口类就只在hPrevInstance是NULL的时候才注册,这表明程序运行没有其他实例。

在32位的Windows中,hPrevInstance总是NULL。此代码将始终正常工作,且不必检查hPrevInstance。

3.1.5 创建窗口

窗口类定义了窗口的一般特征,因此可以使用同一窗口类创建许多不同的窗口。实际调用CreateWindow创建窗口时,可能指定有关窗口的更详细的信息。

新的Windows程序员有时会混淆窗口类和窗口之间的区别,以及为什么一个窗口的所有特征不能被一步到位地指定。实际上,以这种方式分开这些风格信息是非常方便的。例如,所有的按钮窗口都可以基于同样的窗口类来创建。与这个窗口类相关的窗口过程位于Windows内部。由窗口类来负责处理按钮的键盘和鼠标输入,并定义按钮在屏幕上的外观形象。从这一点看来,所有的按钮都是以同样的方式工作的。但是并非所有的按钮都是一样的。它们可以有不同的大小,不同的屏幕位置,以及不同的正文串。后面的这样一些特征是窗口定义的一部分,而不是窗口类定义的。

传递给RegisterClass函数的信息在一个数据结构中说明,而传递给CreateWindow函数的信息作为函数的单独参数来说明。下面是HELLOWIN.C中的CreateWindows调用,每一个域都做了完整的注释:

```
hwnd = CreateWindow (szAppName,                  // window class name
                     TEXT ("The Hello Program"), // window caption
                     WS_OVERLAPPEDWINDOW,        // window style
                     CW_USEDEFAULT,              // initial x position
                     CW_USEDEFAULT,              // initial y position
                     CW_USEDEFAULT,              // initial x size
                     CW_USEDEFAULT,              // initial y size
                     NULL,                       // parent window handle
                     NULL,                       // window menu handle
                     hInstance,                  // program instance handle
                     NULL) ;                     // creation parameters
```

这时,我不想提及实际运行的CreateWindowA函数和CreateWindowW函数,是这两个函数分别按ASCII或者Unicode为函数处理前两个参数的。

标记为“window class name”的参数是szAppName,它含有串“HelloWin”——这是程序

注册的窗口类名称。这就是如何把我们正在创建的窗口与一个窗口相联结类的方式。

此程序创建的窗口是一个普通的重叠式窗口。它含有一个标题栏，标题栏左边有一个系统菜单框，标题栏右边有缩小、放大和关闭图标，四周还有一个表示窗口大小的边框。这是标准风格的窗口，名为 WS_OVERLAPPEDWINDOW，该参数按“窗口风格”出现在 CreateWindow 中。如果看一下 WINUSER.H，您将会发现此风格是几种位标识的综合：

```
#define WS_OVERLAPPEDWINDOW (WS_OVERLAPPED     | \
                             WS_CAPTION        | \
                             WS_SYSMENU        | \
                             WS_THICKFRAME     | \
                             WS_MINIMIZEBOX    | \
                             WS_MAXIMIZEBOX)
```

“窗口标题”是显示在标题栏中的文本。

注释为“初始 x 位置”和“初始 y 位置”的参数指定了窗口左上角相对于屏幕左上角的初始位置。由于这些参数使用 CW_USEDEFAULT 标识符，指示 Windows 使用重叠窗口的默认位置（CW_USEDEFAULT 定义为 0x80000000）。默认情况下，Windows 依次对新创建的窗口定位，使各窗口左上角的垂直和水平距离在屏幕上按一定的大小递增。与此类似，“初始 x 大小”和“初始 y 大小”参数分别指定窗口的宽度和高度。同样使用了 CW_USEDEFAULT 标识符，表明希望 Windows 使用默认尺寸。

当创建一个“最高级”窗口，如应用程序窗口时，注释为“父窗口句柄”的参数设置为 NULL。通常，如果有窗口之间存在父子关系，则子窗口总是出现在父窗口的上面。应用程序窗口出现在桌面窗口的上面，但不必为调用 CreateWindow 而找出桌面窗口的句柄。

因为窗口没有菜单，所以“窗口菜单句柄”也设置为 NULL。“程序实例句柄”设置为实例句柄，它是作为 WinMain 的参数传递给这个程序的。最后，“创建参数”指针设置为 NULL，可以用这个指针访问以后想要引用的程序中的数据。

CreateWindow 调用返回被创建的窗口的句柄，该句柄存放在变量 hwnd 中，后者被定义为 HWND 类型（“窗口句柄类型”）。Windows 中的每个窗口都有一个句柄，程序用句柄来引用窗口。许多 Windows 函数需要使用 hwnd 作为参数，这样，Windows 才能知道函数是针对哪个窗口的。如果一个程序创建了许多窗口，则每个窗口均有一个句柄。窗口句柄是 Windows 程序处理的最重要的句柄之一。

3.1.6 显示窗口

在 CreateWindow 调用返回之后，Window 内部已经创建了这个窗口。这就是说，Windows

已经分配了一块内存,用来保存关于在 CreateWindow 调用中指定窗口的全部信息,再加上 Windows 在随后所能找到的所有其他信息。

然而,窗口并未在视频显示器上出现,这就需要再来两个调用,一个是:

```
ShowWindow (hwnd, iCmdShow) ;
```

第一个参数是刚刚用 CreateWindow 创建的窗口句柄。第二个参数是作为参数传给 WinMain 的 iCmdShow。它确定最初如何在屏幕上显示窗口,是常规、最小化还是最大化。在向 Start 菜单添加程序时,用户可能做出最优选择。如果窗口按常规显示,那么从 WinMain 接收并传递给 ShowWindow 的就是 SW_SHOWNORMAL;如果窗口按最大化显示,则为 SW_SHOWMAXIMIZED;而如果窗口只显示在任务栏上,则是 SW_SHOWMINNOACTIVE。

ShowWindow 函数在显示器上显示窗口。如果 ShowWindow 的第二个参数是 SW_SHOWNORMAL,则窗口的客户区域就被窗口类中定义的背景刷子所覆盖。函数调用

```
UpdateWindow (hwnd) ;
```

导致客户区域被绘制。它通过给窗口过程(即 HELLOWIN.C 中的 WndProc 函数)发送一个 WM_PAINT 消息来做到这一点。后面,我们将说明 WndProc 如何处理这个消息。

3.1.7 消息循环

调用 UpdateWindow 之后,窗口就出现在视频显示器上。程序现在必须准备读入用户用键盘和鼠标输入的数据。Windows 为当前运行的每个 Windows 程序维护一个"消息队列"。在发生输入事件之后,Windows 将事件转换为一个"消息",并将消息放入程序的消息队列中。

程序通过执行一块被称为"消息循环"的代码从消息队列中取出消息:

```
while (GetMessage (&msg, NULL, 0, 0))
{    TranslateMessage (&msg) ;
     DispatchMessage (&msg) ;
}
```

msg 变量是类型为 MSG 的结构,类型 MSG 在 WINUSER.H 中定义如下:

```
typedef struct tagMSG
{
     HWND    hwnd ;
     UINT    message ;
     WPARAM  wParam ;
     LPARAM  lParam ;
     DWORD   time ;
     POINT   pt ;
```

```
}
MSG, * PMSG ;
```

POINT 数据类型也是一个结构,它在 WINDEF.H 中定义如下:

```
typedef struct tagPOINT
{
     LONG   x ;
     LONG   y ;
}
POINT, * PPOINT;
```

消息循环以 GetMessage 调用开始,它从消息队列中取出一个消息:

```
GetMessage (&msg, NULL, 0, 0)
```

这一调用传给 Windows 一个指向名为 msg 的 MSG 结构的指针。第二、第三和第四个参数设置为 NULL 或者 0,表示程序接收它自己创建的所有窗口的所有消息。Windows 用从消息队列中取出的下一个消息来填充消息结构的各个域,结构的各个域包括:

- hwnd　消息发向的窗口的句柄。在 HELLOWIN 程序中,这一参数与 CreateWindow 返回的 hwnd 值相同,因为这是该程序拥有的唯一窗口。
- message　消息标识符。这是一个数值,用以标识消息。对于每个消息,均有一个对应的标识符,这些标识符定义于 Windows 头文件(其中大多数在 WINUSER.H 中),以前缀 WM(“window message”,窗口消息)开头。例如,用户将鼠标光标放在 HELLOWIN 客户区域之内,并按下鼠标左键,Windows 就在消息队列中放入一个消息,该消息的 message 域等于 WM_LBUTTONDOWN。这是一个常量,其值为 0x0201。
- wParam　一个 32 位的“message parameter(消息参数)”,其含义和数值根据消息的不同而不同。
- lParam　一个 32 位的消息参数,其值与消息有关。
- time　消息放入消息队列中的时间。
- pt　消息放入消息队列时的鼠标坐标。

只要从消息队列中取出消息的 message 域不为 WM_QUIT(其值为 0x0012),GetMessage 就返回一个非 0 值。WM_QUIT 消息将导致 GetMessage 返回 0。

语句

```
TranslateMessage (&msg) ;
```

将 msg 结构传给 Windows,进行一些键盘转换(关于这一点,我们将在第六章中深入讨论)。语句

```
DispatchMessage (&msg) ;
```

又将 msg 结构回传给 Windows。然后，Windows 将该消息发送给适当的窗口过程，让它进行处理。这也就是说，Windows 将调用窗口过程。在 HELLOWIN 中，这个窗口过程就是 WndPro 函数。处理完消息之后，WndProc 返回到 Windows。此时，Windows 还停留在 DispatchMessage 调用中。在结束 DispatchMessage 调用的处理之后，Windows 回到 HELLOWIN，并且接着从下一个 GetMessage 调用开始消息循环。

3.1.8 窗口过程

以上我们所讨论的实在只是准备性的工作：注册窗口类，创建窗口，然后在屏幕上显示窗口，程序进入消息循环，从消息队列中取出一条消息。

实际的动作发生在窗口过程中。窗口过程确定了在窗口的客户区域中显示些什么，以及窗口怎样响应用户输入。

在 HELLOWIN 中，窗口过程是命名为 WndProc 的函数。窗口过程可任意命名（只要不和其他名字发生冲突）。一个 Windows 程序可以包含多个窗口过程。一个窗口过程总是与调用 RegisterClass 注册的特定窗口类相关联。CreateWindow 函数根据特定窗口类创建一个窗口。但基于一个窗口类，可以创建多个窗口。

窗口过程总是定义为如下形式：

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
```

注意，窗口过程的 4 个参数与 MSG 结构的前 4 个域是相同的。第一个参数 hwnd 是接收消息的窗口的句柄，它与 CreateWindow 函数的返回值相同。对于与 HELLOWIN 相似的程序（只创建一个窗口），这个参数是程序所知道的唯一窗口句柄。如果程序是基于同一窗口类（同时也是同一窗口过程）创建多个窗口，则 hwnd 标识接收该消息的特定窗口。

第二个参数与 MSG 结构中的 message 域相同，它是标识该消息的数字。最后两个参数都是 32 位的消息参数，它提供关于消息的更多信息。这些参数包含每个消息类型的详细信息。有时消息参数是两个存放在一起的 16 位值，而有时消息参数又是一个指向正文串或数据结构的指针。

程序通常不直接调用窗口过程。窗口过程通常由 Windows 本身调用。通过调用 SendMessage 函数，程序能够直接调用它自己的窗口过程。我们将在后面的章节讨论 SendMessage 函数。

3.1.9 处理消息

窗口过程接收的每个消息均是用一个数值来标识的，也就是传给窗口过程的 message 参数。Windows 头文件 WINUSER.H 为每个消息参数定义以“WM”（窗口消息）为前缀的标

识符。

一般来说，Windows 程序员使用 switch 和 case 结构来确定窗口过程接收的是什么消息，以及如何适当地处理它。窗口过程在处理消息时，必须返回 0。窗口过程不予处理的所有消息应该被传给名为 DefWindowProc 的 Windows 函数。从 DefWindowProc 返回的值必须由窗口过程返回。

在 HELLOWIN 中，WndProc 只选择处理三种消息：WM_CREATE、WM_PAINT 和 WM_DESTROY。窗口过程的结构如下：

```
switch (iMsg)
{
case WM_CREATE :
    [process WM_CREATE message]
    return 0 ;

case WM_PAINT :
    [process WM_PAINT message]
    return 0 ;

case WM_DESTROY :
    [process WM_DESTROY message]
    return 0 ;
}
return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
```

调用 DefWindowProc 来为窗口过程不予处理的所有消息提供默认处理，这是很重要的。除非能按常规运行，比如能在不工作时终止程序。

3.1.10 播放声音文件

窗口过程接收的第一个消息——也是 WndProc 选择处理的第一个消息——是 WM_CREATE。当 Windows 在 WinMain 中处理 CreateWindow 函数时，WndProc 接收这个消息。即当 HELLOWIN 调用 CreateWindow 时，Windows 将做一些它必须做的工作。在这些工作中，Windows 调用 WndProc，将第一个参数设置为窗口句柄，第二个参数设置为 WM_CREATE。WndProc 处理 WM_CREATE 消息，并将控制返回给 Windows。Windows 然后可以从 CreateWindow 调用中返回到 HELLOWIN，继续在 WinMain 中进行下一步的处理。

通常，窗口过程在 WM_CREATE 处理期间进行一次窗口初始化。HELLOWIN 对这条消息的处理中播放一个名为 HELLOWIN.WAV 的声音文件。它使用简单的 PlaySound 函数来做到这一点。该函数说明在/Platform SDK/Graphics and Multimedia Services/Multimedia Audio/Waveform Audio 中，而文档在/Platform SDK/Graphics and Multimedia Services/Multimedia Reference/Multimedia Functions 中。

PlaySound 的第一个参数是声音文件的名称。(它也可能是在 Control Panel 的 Sounds 中定义的一种声音的别名,或者是一个程序资源。)第二个参数只有当声音文件是一种资源时才被使用。第三个参数指定一些选项。在这个例子中,指定第一个参数是一个文件名,并且异步地播放声音,即 PlaySound 函数调用在声音文件开始播放时立即返回,而不会等待它的完成。在这种方法下,程序能够继续初始化。

WndProc 通过从窗口过程中返回 0,结束整个 WM_CREATE 的处理。

3.1.11 WM_PAINT 消息

WndProc 处理的第二个消息为 WM_PAINT。这个消息在 Windows 程序设计中是很重要的。当窗口客户区域的一部分或者全部变为"无效",以致于必须"刷新"时,将由这个消息通知程序。

客户区域怎么会变得无效呢?在最初创建窗口的时候,整个客户区域都是无效的,因为程序还没有在窗口上画什么东西。第一个 WM_PAINT 消息(通常发生在 WinMain 中调用 UpdateWindow 时)指示窗口过程在客户区域上画一些东西。

在用户改变 HELLOWIN 窗口的大小后,客户区域重新变得无效。读者应该还记得,HELLOWIN 中 wndclass 结构的 style 域设置为标志 CS_HREDRAW 和 CS_VREDRAW,这一风格指示 Windows,在窗口大小改变后,使整个窗口无效。然后,窗口过程将收到一个 WM_PAINT 消息。

当用户将 HELLOWIN 最小化,然后再次将窗口恢复为以前的大小时,Windows 将不会保存客户区域的内容。在图形环境下,这涉及的数据量很大。因此,Windows 令窗口无效,窗口过程接收一个 WM_PAINT 消息,并自动恢复其窗口的内容。

在移动窗口以使其相互重叠时,Windows 不保存一个窗口中被另一个窗口所遮盖的内容。在这一部分不再被遮盖之后,它就被标志为无效。窗口过程接收到一个 WM_PAINT 消息,以刷新窗口的内容。

对 WM_PAINT 的处理几乎总是从一个 BeginPaint 调用开始:

```
hdc = BeginPaint (hwnd, &ps) ;
```

而以一个 EndPaint 调用结束:

```
EndPaint (hwnd, &ps) ;
```

在这两个调用中,第一个参数都是程序的窗口句柄,第二个参数是指向类型为 PAINTSTRUCT 的结构指针。PAINTSTRUCT 结构中包含一些窗口过程,可以用来刷新客户区域的内容。我们将在下一章中讨论该结构的各个域。现在我们只在 BeginPaint 和 EndPaint 函数中用到它。

在 BeginPaint 调用中,如果客户区域的背景还未被删除,则由 Windows 来删除。它使用注册窗口类的 WNDCLASS 结构的 hbrBackground 域中指定的刷子来删除背景。在 HELLOWIN 中，这是一个白色备用刷子。这意味着,Windows 将通过把窗口背景设置为白色来删除窗口背景。BeginPaint 调用使整个客户区域有效,并返回一个“设备描述表句柄”。设备描述表是指物理输出设备(如视频显示器)及其设备驱动程序。在窗口的客户区域显示文本和图形需要设备描述表句柄。但是不能用从 BeginPaint 返回的设备描述表句柄在客户区域之外绘图,读者可以试一试。EndPaint 释放设备描述表句柄,使之不再有效。

如果窗口过程不处理 WM_PAINT 消息(这是非常少见的情况),它们必须被传送给 DefWindowProc。DefWindowProc 只是依次调用 BeginPaint 和 EndPaint,以使客户区域有效。

调用完 BeginPaint 之后,WndProc 接着调用 GetClientRect:

```
GetClientRect (hwnd, &rect) ;
```

第一个参数是程序窗口的句柄。第二个参数是一个指针,指向一个 RECT 类型的 rectangle 结构。该结构有 4 个 LONG 域,分别为 left、top、right 和 bottom。GetClientRect 将这 4 个域设置为窗口客户区域的尺寸。left 和 top 域通常设置为 0,right 和 bottom 域设置为客户区域的宽度和高度(像素点数)。

WndProc 除了将该 RECT 结构指针作为 DrawText 的第 4 个参数传递外,不再对它做其他处理:

```
DrawText (hdc, TEXT ("Hello, Windows 98!"), -1, &rect,
        DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
```

DrawText 可以输出文本(正如其名字所表明的一样)。由于该函数要输出文本,第一个参数是从 BeginPaint 返回的设备描述表句柄,第二个参数是要输出的文本,第三个参数是 -1,指示文本串是以字节 0 终结的。

DrawText 最后一个参数是一系列位标志,它们均在 WINUSER.H 中定义。(虽然由于其显示输出的效果,使得 DrawText 像一个 GDI 函数调用,但它确实因为相当高级的画图功能而成为用户模式的一部分。此函数在/Platform SDK/Graphics and Multimedia Services/GDI/Fonts and Text 中说明。)标志指示了文本必须显示在一行上,水平方向和垂直方向都位于第 4 个参数指定的矩形中央。因此,这个函数调用将导致串“Hello, Windows 98!”显示在客户区域的中央。

一旦客户区域变得无效(正如在改变大小时所发生的情况一样),WndProc 就接收一个新的 WM_PAINT 消息。WndProc 通过调用 GetClientRect 获取变化后的窗口大小,并在新窗口的中央显示文本。

3.1.12 WM_DESTROY 消息

WM_DESTROY 消息是另一个重要消息。这一个消息指示，Windows 正在根据用户输入的命令来清除窗口。该消息是用户单击 Close 按钮或者在程序的系统菜单上选择 Close 时发生的。（在本章的后面，我们将详细讨论 WM_DESTROY 消息是如何生效的。）

HELLOWIN 通过调用 PostQuitMessage 以标准方式响应 WM_DESTROY 消息：

```
PostQuitMessage (0) ;
```

该函数在程序的消息队列中插入一个 WM_QUIT 消息。前面提到过，GetMessage 对于除了 WM_QUIT 之外的从消息队列中取出的所有消息都返回非 0 值。而当 GetMessage 取到一个 WM_QUIT 消息时，它返回 0。这将导致 WinMain 退出消息循环，并终止程序。然后程序执行下面的语句：

```
return msg.wParam ;
```

结构的 wParam 域是传递给 PostQuitMessage 函数的值（通常是 0）。然后返回语句将退出 WinMain 并终止程序。

3.2 Windows 编程的难点

即使有了对 HELLOWIN 的说明，读者可能仍然对程序的结构和原理觉得神秘。在为传统环境编写简单的 C 程序时，整个程序可能包含在 main 函数中。而在 HELLOWIN 中，WinMain 只包含了注册窗口类，创建窗口，从消息队列中取出消息和发送消息所必需的代码。

程序的所有实际动作均在窗口过程中发生。在 HELLOWIN 中，这些动作不多，WndProc 只是简单地播放了一个声音文件，并在窗口中显示一个文本串。但是在后面的章节中，读者将发现，Windows 程序所做的一切都是响应发送给窗口过程的消息。这是概念上的主要难点之一，在开始编写 Windows 程序之前，必须先搞清楚。

3.2.1 别调用我，我会调用您

前面我们提到过，程序员已经熟悉了使用操作系统调用的思路。例如，C 程序员使用 fopen 函数打开文件。fopen 函数最终通过调用操作系统来打开文件。这毫无问题。

但是 Windows 不同，尽管 Windows 有 1000 个以上的函数可供程序调用，但 Windows 也调用用户程序，比如前面定义的窗口过程 WndProc。窗口过程与一个窗口类相关联，窗口类是程序调用 RegisterClass 注册的。基于该类创建的窗口使用这个窗口过程来处

理窗口的所有消息。Windows 通过调用窗口过程来给窗口发送一个消息。

在第一次创建窗口时，Windows 调用 WndProc。在窗口清除时，Windows 也调用 WndProc。窗口改变大小、移动或者变成图标时，从菜单中选择某一项、滚动滚动条、按下鼠标键或者从键盘输入字符时，以及窗口客户区域必须被刷新时，Windows 都要调用 WndProc。

所有这些 WndProc 调用都以消息的形式进行。在大多数 Windows 程序中，程序的主要部分都用来处理消息。Windows 可以发送给窗口过程的消息通常都以 WM 打头的名字标识，并且都在 WINUSER.H 头文件中定义。

实际上，从程序外调用程序内的例程这一思想，在传统的程序设计中并非闻所未闻。C 中的 signal 函数可以捕获 Ctrl-C 中断或操作系统的其他中断。为 MS-DOS 编写的老程序中经常有捕获硬件中断的语句。

但在 Windows 中，这种概念扩展为包括一切事件。窗口中发生的一切都以消息的形式传给窗口过程。然后，窗口过程以某种方式响应这个消息，或者将消息传给 DefWindowProc，进行默认处理。

在 HELLOWIN 中，窗口过程的 wParam 和 lParam 参数除了作为传递给 DefWindowProc 的参数外，不再有其他用处。这些参数给出了关于消息的其他信息，参数的含义与具体消息相关。

让我们来看一个例子。一旦窗口的客户区域大小发生了改变，Windows 就调用窗口的窗口过程。窗口过程的 hwnd 参数是改变大小的窗口的句柄。（请记住，一个窗口过程能处理基于同一个窗口类创建的多个窗口的消息。参数 hwnd 让窗口过程知道是哪个窗口在接收消息。）参数 message 是 WM_SIZE。消息 WM_SIZE 的参数 wParam 的值是 SIZE_RESTORED、SIZE_MINIMIZED、SIZE_MAXIMIZED、SIZE_MAXSHOW 或 SIZE_MAXHIDE（在 WINUSER.H 头文件中分别定义为数字 0 到 4）。也就是说，参数 wParam 表明窗口是非最小化还是非最大化，是最小化、最大化，还是隐藏。

lParam 参数包含了新窗口的大小，新宽度和新高度均为 16 位值，合在一起成为 32 位的 lParam。WINDEF.H 中提供了帮助程序员从 lParam 中取出这两个值的宏，我们将在下一章说明这个宏。

有时候，DefWindowProc 处理完消息后会产生其他的消息。例如，假设用户运行 HELLOWIN，并且用户最终单击了关闭按钮，或者假设用键盘或鼠标从系统菜单中选择了 Close，DefWindowProc 处理这一键盘或者鼠标输入，在检测到用户选择了 CLOSE 选项之后，它给窗口过程发送一个 WM_SYSCOMMAND 消息。WndProc 将这个消息传给 DefWindowProc。DefWindowProc 给窗口过程发送一个 WM_CLOSE 消息来响应之。WndProc 再次将它传给 DefWindowProc。DestroyWindow 调用 DestroyWindow 来响应这个 WM_CLOSE 消息。DestroyWindow 导致 Windows 给窗口过程发送一个 WM_DESTROY 消息。

WndProc 再调用 PostQuitMessage，将一个 WM_QUIT 消息放入消息队列中，以此来响应此消息。这个消息导致 WinMain 中的消息循环终止，然后程序结束。

3.2.2 进队消息与不进队消息

我们已经谈到过，Windows 给窗口发送消息，这意味着 Windows 调用窗口过程。但是，Windows 程序也有一个消息循环，它调用 GetMessage 从消息队列中取出消息，并且调用 DispatchMessage 将消息发送给窗口过程。

那么，Windows 程序是依次等待消息（类似于普通程序中等价的键盘输入），然后将消息送到某地方去的吗？或者，它是直接从程序外面接收消息的吗？实际上，两种情况都存在。

消息能够被分为“进队的”和“不进队的”。进队的消息是由 Windows 放入程序消息队列中的。在程序的消息循环中，重新返回并分配给窗口过程。不进队的消息在 Windows 调用窗口时直接送给窗口过程。也就是说，进队的消息被“发送”给消息队列，而不进队的消息则“发送”给窗口过程。任何情况下，窗口过程都将获得窗口所有的消息——包括进队的和不进队的。窗口过程是窗口的“消息中心”。

进队消息基本上是用户输入的结果，以击键（如 WM_KEYDOWN 和 WM_KEYUP 消息）、击键产生的字符（WM_CHAR）、鼠标移动（WM_MOUSEMOVE）和鼠标键（WM_LBUTTONDOWN）的形式给出。进队消息还包含时钟消息（WM_TIMER）、刷新消息（WM_PAINT）和退出消息（WM_QUIT）。

不进队消息则是其他消息。在许多情况下，不进队消息来自调用特定的 Windows 函数。例如，当 WinMain 调用 CreateWindow 时，Windows 将创建窗口并在处理中给窗口过程发送一个 WM_CREATE 消息。当 WinMain 调用 ShowWindow 时，Windows 将给窗口过程发送 WM_SIZE 和 WM_SHOWWINDOW 消息。当 WinMain 调用 UpdateWindow 时，Windows 将给窗口过程发送 WM_PAINT 消息。键盘或鼠标输入时发出的进队消息信号，也能在不进队消息中出现。例如，用键盘或鼠标选择了一个菜单项时，键盘或鼠标消息就是进队的，而说明菜单项已选中的 WM_COMMAND 消息则可能就是不进队的。

这一过程显然很复杂，但幸运的是，其中的大部分是由 Windows 解决的，不关我们的程序的事。从窗口过程的角度来看，这些消息是以一种有序的、同步的方式进出的。窗口过程可以处理它们，也可以不处理。

当我说消息是以一种有序的同步的方式进出时，我是说首先消息与硬件的中断不同。在一个窗口过程中处理消息时，程序不会被其他消息突然中断。

虽然 Windows 程序可以多线程执行，但每个线程的消息队列只为窗口过程在该线程中执行的窗口处理消息。换句话说，消息循环和窗口过程不是并发运行的。当一个

消息循环从其消息队列中接收一个消息，然后调用 DispatchMessage 将消息发送给窗口过程时，直到窗口过程将控制返回给 Windows，DispatchMessage 才能返回。

当然，窗口过程能调用给窗口过程发送另一个消息的函数。这时，窗口过程必须在函数调用返回之前完成对第二个消息的处理。那时窗口过程将处理最初的消息。例如，当窗口过程调用 UpdateWindow 时，Windows 将调用窗口过程来处理 WM_PAINT 消息。窗口过程处理 WM_PAINT 消息结束以后，UpdateWindow 调用将把控制返回给窗口过程。

这也就是说窗口过程必须是可重入的。在大多数情况下，这不会带来问题，但是程序员应该意识到这一点。例如，假设您在窗口过程中处理一个消息时设置了一个静态变量，然后调用了一个 Windows 函数。在这个函数返回时，您还能保证那个变量的值还是原来那个吗？难说——很可能您调用的 Windows 函数产生了另外一个消息，并且窗口过程在处理这条消息时改变了该变量的值。这也是在编译 Windows 程序时，有些编译优化选项必须关闭的原因之一。

在许多情况下，窗口过程必须保存它从消息中获取的信息，并在处理另一个消息时使用这些信息。这些信息可以存储在窗口的 static 变量或全程变量中。

当然，读者将在下面几章对此有一个更清楚的理解，因为窗口过程将处理更多的消息。

3.2.3 行动迅速

Windows 98 和 Windows NT 都是抢占式的多任务环境。这意味着当一个程序在进行一项长时间工作时，Windows 可以允许用户将控制切换到另一个程序中。这是一件好事，也是当前 Windows 版优于以前 16 位 Windows 的地方。

然而，由于 Windows 构造的方式，这种抢占式多任务并不总是以您希望的方式工作。例如，假设您的程序花费一分钟左右来处理某一个消息。是的，用户可以将控制切换到另一个程序，但是却无法对您的程序进行任何动作。用户无法移动您的程序窗口，缩放它，最小化，关闭它，什么都不能做。这是因为您的窗口过程正忙于进行一项长作业。表面上并不是窗口过程在执行它自己的移动和缩放操作，但实际上确实是它在做。这就是 DefWindowProc 部分的工作，它必须被考虑为您的窗口过程的一部分。

如果您的程序在处理某些消息时需要长作业的话，可以选择我在第二十章里描述的那些方法来做得更有礼貌一些。即使是在抢占式多任务环境中，也不应该让您的程序呆在屏幕上一动不动。这会让用户讨厌的，他们会认为您的程序中有 bug、不标准的动作，或者没有完成的帮助文件。给用户一个中断，然后从全部消息中快速返回。

第四章

输出文本

在前一章，您看到了一个简单的 Windows 98 程序，它在窗口中央，或者更准确地说，在客户区中央显示一行文本。正如我们学到的，客户区是整个应用程序窗口中未被标题栏、窗口边框以及可选的菜单栏、工具栏、状态栏和滚动条占据的部分。简而言之，客户区是窗口中可以由程序任意书写和传递可视信息的部分。

对于程序的客户区，您几乎可以为所欲为，只不过必须认为其大小是某一特定尺寸，或者在程序运行时其大小保持不变。如果您不熟悉图形窗口环境的编程，这些约定可能会使您感到惊讶：不能再考虑 80 个字符一行的固定数字了。您的程序必须与其他 Windows 程序共享视频显示器。Windows 用户控制程序窗口在屏幕上显示的方式。尽管可以创建固定大小的窗口（这对于计算器之类的应用是合理的），但在大多数情况下，用户应该能够改变应用程序窗口的大小。您的程序必须能够接受指定给它的大小，并且合理地利用这一空间。

这有两种可能的情况。一种可能是，程序只有仅能显示“hello”的客户区；还有另一种可能，即程序在一个大屏幕、高分辨率的系统上运行，其客户区大得足以显示两整页文本。灵活地处理这两种极端是 Windows 程序设计的要点之一。

这一章，我们将讲述程序在客户区显示信息的方式，但比上一章说明的显示方式更加复杂。当程序在客户区显示文本或图形时，它经常要“绘制”它的客户区。本章着重讲述绘制的方法。

尽管 Windows 为显示图形提供了强大的图形设备接口（GDI）函数，但在这一章中，我只介绍简单文本行的显示。我也将忽略 Windows 能够使用的不同字体外形及字体大小，仅使用 Windows 的默认系统字体。这看起来似乎是一种限制，其实不然，本章涉及和解决的问题适用于所有 Windows 程序设计。在混合显示文本和图形时，Windows 默认字体的字符大小通常决定了图形的尺寸。

本章表面上是讨论绘图的方法，实际上是讨论与设备无关的程序设计基础。Win-

dows 程序只能对客户区大小甚至字符的大小做很少的假定，相反，必须使用 Windows 提供的功能来获取关于程序运行环境的信息。

4.1　绘制和刷新

在字符模式环境中，程序可以在显示器的任意部分输出，程序输出到屏幕上的内容会停留在原处，不会神秘地消失。因此，程序可以丢掉重新生成屏幕显示时所需的信息。

在 Windows 中，只能在窗口的客户区绘制文本和图形，而且不能确保在客户区内显示的内容会一直保留到程序有意地改写它。例如，用户可能会在屏幕上移动另一个程序的窗口，这样就可能覆盖您的应用程序窗口的一部分。Windows 不会保存您的窗口中被其他程序覆盖的区域，当程序移开后，Windows 会要求您的程序刷新客户区的这个部分。

Windows 是一个消息驱动系统。它通过把消息投入应用程序消息队列中或者把消息发送给合适的窗口过程，将发生的各种事件通知给应用程序。Windows 通过发送 WM_PAINT 消息通知窗口过程，窗口的部分客户区需要绘制。

4.1.1　WM_PAINT 消息

大多数 Windows 程序在 WinMain 中进入消息循环之前的初始化期间都要调用函数 UpdateWindow。Windows 利用这个机会给窗口过程发送第一个 WM_PAINT 消息。这个消息通知窗口过程：必须绘制客户区。此后，窗口过程应在任何时刻都准备好处理其他 WM_PAINT 消息，必要的话，甚至重新绘制窗口的整个客户区。在发生下面几种事件之一时，窗口过程会接收到一个 WM_PAINT 消息：

- 在用户移动窗口或显示窗口时，窗口中先前被隐藏的区域重新可见；
- 用户改变窗口的大小（如果窗口类风格具有 CS_HREDRAW 和 CS_VREDRAW 位设置）；
- 程序使用 ScrollWindow 或 ScrollDC 函数滚动客户区的一部分；
- 程序使用 InvalidateRect 或 InvalidateRgn 函数显式产生 WM_PAINT 消息。

在某些情况下，客户区的一部分被临时覆盖，Windows 试图保存一个显示区域，并在以后恢复它，但这并不总是能成功的。下面的情况下，Windows 可能发送 WM_PAINT 消息：

- Windows 擦除覆盖了部分窗口的对话框或消息框；
- 菜单下拉出来，然后被释放；
- 显示工具提示。

在某些情况下，Windows 总是保存它所覆盖的显示区域，然后恢复它。这些情况是：

- 鼠标光标穿越客户区；
- 图标拖过客户区。

处理 WM_PAINT 消息要求程序员改变自己向显示器输出的思维方式。程序应该组织成可以保留绘制客户区需要的所有信息，并且仅当"有需求"——即 Windows 给窗口过程发送 WM_PAINT 消息时才进行绘制。如果程序在其他时间需要更新其客户区，它可以强制 Windows 产生一个 WM_PAINT 消息。这看来似乎是在屏幕上显示内容的一种舍近求远的方法。但您的程序结构可以从中受益。

4.1.2 有效矩形和无效矩形

尽管窗口过程一旦接收到 WM_PAINT 消息之后，就准备更新整个客户区，但它经常只需要更新一个较小的区域(最常见的是客户区中的矩形区域)。显然，当对话框覆盖了部分客户区时，情况即是如此。在擦除对话框之后，需要重画的只是先前被对话框遮住的矩形区域。

这个区域称为"无效区域"或"更新区域"。正是客户区内无效区域的存在，才提示了 Windows 将一个 WM_PAINT 消息放在应用程序的消息队列中。只有在客户区的某一部分失效时，窗口才会接收 WM_PAINT 消息。

Windows 内部为每个窗口保存一个"绘图信息结构"，这个结构包含了包围无效区域的最小矩形的坐标以及其他信息，这个矩形就叫做"无效矩形"，有时也称为"无效区域"。如果在窗口过程处理 WM_PAINT 消息之前客户区中的另一个区域变为无效，则 Windows 计算出一个包围两个区域的新的无效区域(以及一个新的无效矩形)，并将这种变化后的信息放在绘制信息结构中。Windows 不会将多个 WM_PAINT 消息都放在消息队列中。

窗口过程可以通过调用 InvalidateRect 使客户区内的矩形无效。如果消息队列中已经包含一个 WM_PAINT 消息，Windows 将计算出新的无效矩形；否则，它将一个新的 WM_PAINT 消息放入消息队列中。在接收到 WM_PAINT 消息时，窗口过程可以获取无效矩形的坐标(我们马上就会看到这一点)。通过调用 GetUpdateRect，可以在任何时候获取这些坐标。

在处理 WM_PAINT 消息期间，窗口过程在调用了 BeginPaint 之后，整个客户区即变为有效。程序也可以通过调用 ValidateRect 函数使客户区内的任意矩形区域变为有效。如果这条调用具有令整个无效区域变为有效的效果，则当前队列中的任意 WM_PAINT

消息都将被删除。

4.2 GDI 简介

要在窗口的客户区绘图，可以使用 Windows 的图形设备接口(GDI)函数。Windows 提供了几个 GDI 函数，用于将文本串输出到窗口的客户区内。我们已经在上一章遇到过 DrawText 函数，但是目前使用最为普遍的文本输出函数是 TextOut。该函数的格式如下：

```
TextOut (hdc, x, y, psText, iLength) ;
```

TextOut 向窗口的客户区写入字符串。psText 参数是指向字符串的指针，iLength 是字符串的长度。x 和 y 参数定义了字符串在客户区的开始位置(不久会讲述关于它们的详细情况)。hdc 参数是“设备描述表句柄”，它是 GDI 的重要部分。实际上，每个 GDI 函数都需要将这个句柄作为函数的第一个参数。

4.2.1 设备描述表

读者可能还记得，句柄只不过是一个数值，Windows 用它在内部引用对象。程序员从 Windows 获取句柄，然后在其他函数中使用该句柄。设备描述表句柄是 GDI 函数的窗口“密码”，有了这种设备描述表句柄，程序员就能自如地在客户区上绘图，使图形如自己所愿地变得好看或者难看。

设备描述表(简称为“DC”)实际上是 GDI 内部保存的数据结构。设备描述表与特定的显示设备(如视频显示器或打印机)相关。对于视频显示器，设备描述表总是与显示器上的特定窗口相关。

设备描述表中的有些值是图形化的“属性”，这些属性定义了一些 GDI 绘图函数工作情况的特殊内容。例如，对于 TextOut，设备描述表的属性确定了文本的颜色、文本的背景色、TextOut 函数的 x 坐标和 y 坐标映射到窗口的客户区的方式，以及显示文本时 Windows 使用的字体。

当程序需要绘图时，它必须先获取设备描述表句柄。在获取了该句柄后，Windows 用默认的属性值填充内部设备描述表结构。在后面的章节中您会看到，可以通过调用不同的 GDI 函数改变这些默认值。利用其他的 GDI 函数可以获取这些属性的当前值。当然，还有其他的 GDI 函数能够在窗口的客户区真正地绘图。

当程序在客户区绘图完毕后，它必须释放设备描述表句柄。句柄被程序释放后就不再有效，且不能再被使用。程序必须在处理单个消息期间获取和释放句柄。除了调用 CreateDC(函数，在本章不讲述)创建的设备描述表之外，程序不能在两个消息之间保存其他设备描述表句柄。

Windows 应用程序一般使用两种方法来获取设备描述表句柄，以备在屏幕上绘图。

4.2.2 获取设备描述表句柄：方法一

在处理 WM_PAINT 消息时，使用这种方法。它涉及 BeginPaint 和 EndPaint 两个函数，这两个函数需要窗口句柄(作为参数传给窗口过程)和 PAINTSTRUCT 结构的变量(在 WINUSER.H 头文件中定义)的地址为参数。Windows 程序员通常把这一结构变量命名为 ps，并且在窗口过程中定义它：

```
PAINTSTRUCT ps ;
```

在处理 WM_PAINT 消息时，窗口过程首先调用 BeginPaint。BeginPaint 函数一般在准备绘制时导致无效区域的背景被擦除。该函数也填入 ps 结构的域。BeginPaint 返回的值是设备描述表句柄，这一返回值通常被保存在被叫做 hdc 的变量中。它在窗口过程中的定义如下：

```
HDC hdc ;
```

HDC 数据类型定义为 32 位的无符号整数。然后，程序就可以使用需要设备描述表句柄的 TextOut 等 GDI 函数。调用 EndPaint 即可释放设备描述表句柄。

一般地，处理 WM_PAINT 消息的形式如下：

```
case WM_PAINT:
    hdc = BeginPaint (hwnd, &ps) ;
          [ use GDI functions ]
    EndPaint (hwnd, &ps) ;
    return 0 ;
```

在处理 WM_PAINT 消息时，必须成对地调用 BeginPaint 和 EndPaint。如果窗口过程不处理 WM_PAINT 消息，则它必须将 WM_PAINT 消息传递给 Windows 中 DefWindowProc(默认窗口过程)。DefWindowProc 以下列代码处理 WM_PAINT 消息：

```
case WM_PAINT:
    BeginPaint (hwnd, &ps) ;
    EndPaint (hwnd, &ps) ;
    return 0 ;
```

这一 BeginPaint 和 EndPaint 调用序列中没有任何语句，仅仅使先前无效区域变为有效。

但以下方法是错误的：

```
case WM_PAINT:
    return 0 ;    // WRONG !!!
```

Windows 将一个 WM_PAINT 消息放到消息队列中，是因为客户区的一部分无效。如果不调用 BeginPaint 和 EndPaint(或者 ValidateRect)，则 Windows 不会使该区域变为有效。

相反，Windows 将发送另一个 WM_PAINT 消息，并且会一直发送下去。

4.2.3 绘图信息结构

前面提到过，Windows 为每个窗口保存一个“绘图信息结构”，这就是 PAINTSTRUCT，定义如下：

```
typedef struct tagPAINTSTRUCT
{
     HDC       hdc ;
     BOOL      fErase ;
     RECT      rcPaint ;
     BOOL      fRestore ;
     BOOL      fIncUpdate ;
     BYTE      rgbReserved[32] ;
} PAINTSTRUCT ;
```

在程序调用 BeginPaint 时，Windows 填充该结构的各个字段。用户程序只使用前三个字段，其他字段由 Windows 内部使用。hdc 字段是设备描述表句柄。在旧版本的 Windows 中，BeginPaint 的返回值也曾是这个设备描述表句柄。在大多数情况下，fErase 被标志为 FALSE(0)，这意味着 Windows 已经擦除了无效矩形的背景。这最早在 BeginPaint 函数中发生。(如果要在窗口过程中自己定义一些背景的擦除，可以处理 WM_ERASEBKGND 消息。)Windows 使用 WNDCLASS 结构的 hbrBackground 字段指定的刷子来擦除背景，这个 WNDCLASS 结构是程序在 WinMain 初始化期间登录窗口类时使用的。许多 Windows 程序使用白色刷子。以下显示了程序设置窗口类结构的字段的语句：

```
wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
```

不过，如果程序通过调用 Windows 函数 InvalidateRect 使客户区中的矩形失效，则该函数的最后一个参数会指定是否擦除背景。如果这个参数为 FALSE(即 0)，则 Windows 将不会擦除背景，并且在调用完 BeginPaint 后 PAINTSTRUCT 结构的 fErase 域将为 TRUE (非零)。

PAINTSTRUCT 结构的 rcPaint 域是 RECT 类型的结构。您已经在第三章中看到，RECT 结构定义了一个矩形，其 4 个字段为 left、top、right 和 bottom。PAINTSTRUCT 结构的 rcPaint 字段定义了无效矩形的边界，如下页图 4-1 所示。这些值均以像素点为单位，并相对于客户区的左上角。无效矩形是应该重画的区域。PAINTSTRUCT 中的 rcPaint 矩形不仅是无效矩形，它还是一个“剪取”矩形。这意味着 Windows 将绘图操作限制在剪取矩形内(更确切地说，如果无效矩形区域不为矩形，则 Windows 将绘图操作限制在这个区域内)。

要在处理 WM_PAINT 消息时在更新的矩形外绘图，您可以在调用 BeginPaint 之前

使用如下调用:

```
InvalidateRect(hwnd, NULL, TRUE);
```

它使整个客户区变为无效,并擦除背景。但是,如果最后一个参数等于FALSE,则不擦除背景,原有的东西将保留在原处。

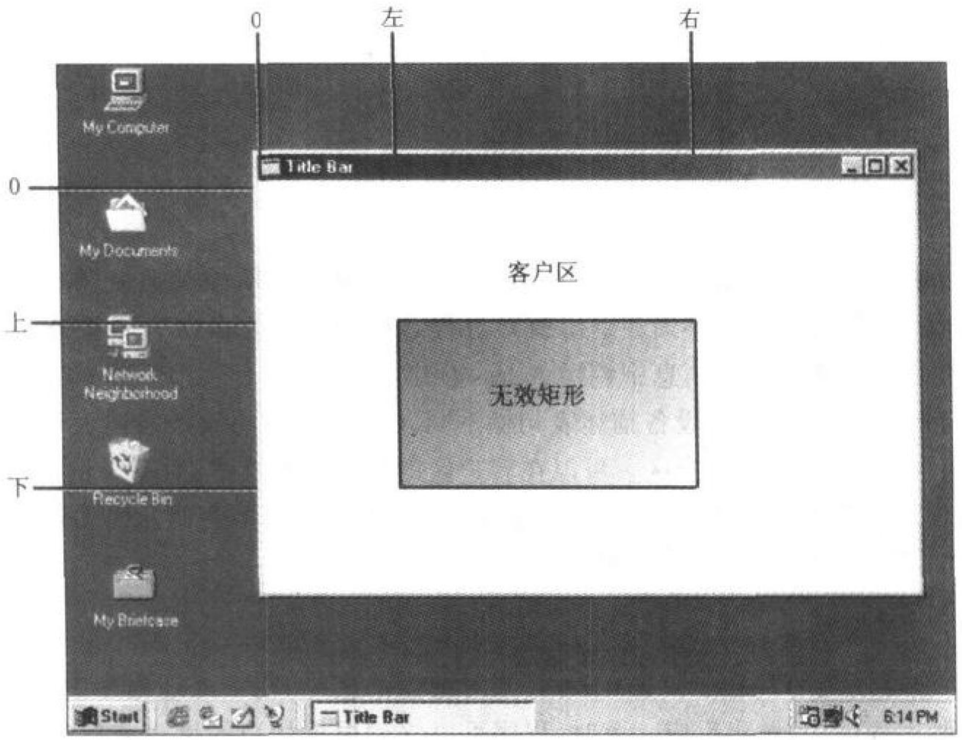


图 4-1 无效矩形的边界

通常这是Windows程序在无论何时收到WM_PAINT消息而不考虑rcPaint结构的情况下简单地重画整个客户区的最方便的方法。例如,如果在客户区的显示输出中包括了一个圆,但是只有圆的一部分落到了无效矩形中,它就使仅绘制圆的无效部分变得没有意义。这需要画整个圆。在您使用从BeginPaint返回的设备描述表句柄时,Windows不会绘制rcPaint矩形外的任何部分。

在第二章的HELLOWIN程序中,我们并不关心处理WM_PAINT消息时的无效矩形。如果文本显示区域恰巧在无效矩形内,则由DrawText恢复之。否则,在处理DrawText调用的某个时刻,Windows会确定它无需向显示器上输出。不过,这一决定需要时间。关心程序性能和速度的程序员希望在处理WM_PAINT期间使用无效矩形范围,以避免不必要的GDI调用。如果绘制时需要访问例如位图这样的磁盘文件,则这就显得尤其重要。

4.2.4 获取设备描述表句柄：方法二

虽然最好是在处理 WM_PAINT 消息期间更新整个客户区，但是您也会发现在处理非 WM_PAINT 消息期间绘制客户区的某个部分也是非常有用的。或者您需要将设备描述表句柄用于其他目的，如获取设备描述表的信息。

要得到窗口客户区的设备描述表句柄，可以调用 GetDC 来获取句柄，在使用完后调用 ReleaseDC：

```
hdc = GetDC (hwnd) ;
[use GDI functions]
ReleaseDC (hwnd, hdc) ;
```

与 BeginPaint 和 EndPaint 一样，GetDC 和 ReleaseDC 函数必须成对地使用。如果在处理某消息时调用 GetDC，则必须在退出窗口过程之前调用 ReleaseDC。不要在一个消息中调用 GetDC，却在另一个消息中调用 ReleaseDC。

与从 BeginPaint 返回设备描述表句柄不同，GetDC 返回的设备描述表句柄具有一个剪取矩形，它等于整个客户区。可以在客户区的某一部分绘图，而不只是在无效矩形上绘图(如果确实存在无效矩形)。与 BeginPaint 不同，GetDC 不会使任何无效区域变为有效。如果需要使整个客户区有效，可以调用

```
ValidateRect (hwnd, NULL) ;
```

一般可以调用 GetDC 和 ReleaseDC 来对键盘消息(如在字处理程序中)或鼠标消息(如在画图程序中)做出反应。此时，程序可以立刻根据用户的键盘或鼠标输入来更新客户区，而不需要考虑为了窗口的无效区域而使用 WM_PAINT 消息。不过，一旦确实收到了 WM_PAINT 消息，程序就必须要收集足够的信息后才能更新显示。

与 GetDC 相似的函数是 GetWindowDC。GetDC 返回用于写入窗口客户区的设备描述表句柄，而 GetWindowDC 返回写入整个窗口的设备描述表句柄。例如，您的程序可以使用从 GetWindowDC 返回的设备描述表句柄在窗口的标题栏上写入文字。然而，程序同样也应该处理 WM_NCPAINT (“非客户区绘制”)消息。

4.2.5 TextOut：细节

TextOut 是用于显示文本的最常用的 GDI 函数。语法是：

```
TextOut (hdc, x, y, psText, iLength) ;
```

以下将详细地讨论这个函数。

第一个参数是设备描述表句柄，它既可以是 GetDC 的返回值，也可以是在处理 WM

_PAINT 消息时 BeginPaint 的返回值。

设备描述表的属性控制了被显示的文本串的特征。例如,设备描述表中有一个属性指定文本颜色,默认颜色为黑色;默认设备描述表还定义了白色的背景。在程序向显示器输出文本时,Windows 使用这个背景色来填充字符周围的矩形空间(称为"字符框")。

该文本背景色与定义窗口类时设置的背景并不相同。窗口类中的背景是一个刷子,它是一种纯色或者非纯色组成的刷子,Windows 用它来擦除客户区,它不是设备描述表结构的一部分。在定义窗口类结构时,大多数 Windows 应用程序使用 WHITE_BRUSH,以便默认设备描述表中的默认文本背景颜色与 Windows 用以擦除客户区背景的刷子颜色相同。

psText 参数是指向字符串的指针,iLength 是串中字符的个数。如果 psText 指向 Unicode 字符串,则串中的字节数就是 iLength 值的两倍。串中不能包含任何 ASCII 控制字符(如回车、换行、制表或退格),Windows 会将这些控制字符显示为实心块。TextOut 不识别作为串尾标志的内容为零的字节(对于 Unicode,是一个短整型的 0),而需要 nLength 参数指明长度。

TextOut 中的 x 和 y 定义客户区内字符串的开始位置,x 是水平位置,y 是垂直位置。串中第一个字符的左上角位于坐标点(x,y)。在默认设备描述表中,原点(x 和 y 均为 0 的点)是客户区的左上角。如果在 TextOut 中将 x 和 y 置为 0,则将从客户区左上角开始输出字符串。

当您阅读 GDI 绘图函数的文档时,例如 TextOut,就会发现传递给函数的坐标常常被称为"逻辑坐标"。这意味着什么,我们将在第五章中详细地解释这种情况。现在请注意,Windows 有多种不同的"映射方式",它们用来控制 GDI 绘图函数指定的逻辑坐标转换为显示器的物理像素坐标的方式。映射方式在设备描述表中定义,默认的映射方式是 MM_TEXT(使用 WINGDI.H 头文件中定义的标识符)。在 MM_TEXT 映射方式下,逻辑单位与物理单位相同,都是相对于客户区左上角的像素数;x 的值从左向右递增,y 的值从上向下递增(参见下页图 4-2)。MM_TEXT 坐标系与 Windows 在 PAINTSTRUCT 结构中定义无效矩形时使用的坐标系相同,这为我们带来了很多方便,但是,其他映射方式并非如此。

设备描述表还定义了一个剪取区域。您已经看到,对于从 GetDC 获取的设备描述表句柄,默认的剪取区域是整个客户区;而对于从 BeginPaint 获取的设备描述表句柄,默认的剪取区域则为无效区域。Windows 不会在剪取区域之外的任何位置显示字符串。如果一个字符有一部分在剪取区域外,那么 Windows 将只显示此区域内的那部分内容。要想将输出写到窗口的客户区之外不是那么容易的,所以不用担心会无意识地这样做。

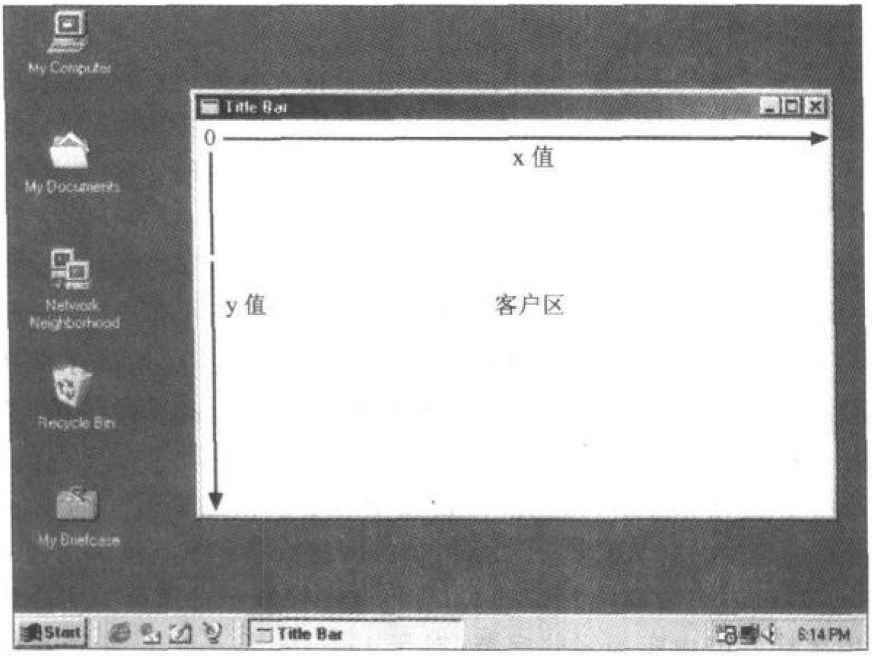


图 4-2　MM_TEXT 映射方式下的 x 坐标和 y 坐标

4.2.6　系统字体

设备描述表还定义了在您调用 TextOut 显示文本时 Windows 使用的字体。默认字体为“系统字体”，或用 Windows 头文件中的标识符即 SYSTEM_FONT。系统字体是 Windows 用来在标题栏、菜单和对话框中显示文本串的默认字体。

在 Windows 的早期版本中，系统字体是等宽(fixed-pitch)字体，这意味着所有字符均具有同样的宽度，非常类似于打字机字体。然而，从 Windows 3.0 开始，系统字体成为一种变宽(variable-pitch)字体，这意味着不同的字符具有不同的大小，比如，“W”要比“i”宽。变宽字体比等宽字体要好读，这已经是公认的事实。不过，可以想象，这一转变使很多原来的 Windows 程序代码不再适用，从而要求程序员学习一些使用字体的新技术。

系统字体是一种“点阵字体”，这意味着字符被定义为像素块。(在第十七章，将讨论 TrueType 字体，它是由轮廓定义的。)对于确定的大小，系统字体的字符大小依赖于视频显示器的大小。系统字体设计为至少能在显示器上显示 25 行、80 列字符文本。

4.2.7　字符大小

要用 TextOut 显示多行文本，就必须确定字体的字符大小，可以根据字符的高度来定位字符的后续行，以及根据字符的宽度来定位字符的后续列。

系统字体的字符高度和平均宽度是多少？这个问题取决于视频显示器的像素大

小。Windows 需要的最小显示大小是 640×480，但是许多用户更喜欢 800×600 或 1024×768 的显示大小。另外，对于这些较大的显示尺寸，Windows 允许用户选择不同大小的系统字体。

程序可以调用 GetSystemMetrics 函数来确定关于用户界面构件大小的信息，调用 GetTextMetrics 来确定字体大小。GetTextMetrics 返回设备描述表中当前选定的字体信息，因此它需要设备描述表句柄。Windows 将文本大小的不同值复制到在 WINGDI.H 中定义的 TEXTMETRIC 类型的结构中。TEXTMETRIC 结构有 20 个字段，我们只使用其中的前 7 个：

```
typedef struct tagTEXTMETRIC
{
     LONG tmHeight ;
     LONG tmAscent ;
     LONG tmDescent ;
     LONG tmInternalLeading ;
     LONG tmExternalLeading ;
     LONG tmAveCharWidth ;
     LONG tmMaxCharWidth ;
          [other structure fields]
}
TEXTMETRIC, * PTEXTMETRIC ;
```

这些字段值的单位取决于选定的设备描述表映射方式。在默认设备描述表下，映射方式是 MM_TEXT，因此值的大小是以像素为单位的。

要使用 GetTextMetrics 函数，需要先定义一个通常被称为 tm 的结构变量：

```
TEXTMETRIC tm ;
```

在需要确定文本尺寸时，先获取设备描述表句柄，再调用 GetTextMetrics：

```
hdc = GetDC (hwnd) ;
GetTextMetrics (hdc, &tm) ;
ReleaseDC (hwnd, hdc) ;
```

此后，您就可以查看文本尺寸结构中的值，并且在可能的情况下，保存其中的一些以备将来使用。

4.2.8 文本尺寸：细节

TEXTMETRIC 结构提供了关于当前设备描述表中选用的字体的丰富信息。但是，字体的纵向大小只由 5 个值确定，其中 4 个值如图 4-3 所示。

最重要的值是 tmHeight，它是 tmAscent 和 tmDescent 的和。这两个值表示了基线上下字符的最大纵向高度。术语“间距(leading)”指打印机在两行文本间插入的空间。在

TEXTMETRIC 结构中，内部的间距包括在 tmAscent 中（因此也在 tmHeight 中），并且它经常是重音符号出现的地方。tmInternalLeading 字段能够被置为 0，在这种情况下，加重音的字母会稍稍缩短，以便容纳重音符号。

TEXTMETRIC 结构还包括一个不包含在 tmHeight 值中的字段 tmExternalLeading。它是字体设计者建议加在横向字符之间的空间大小。在安排文本行之间的空隙时，您可以接受设计者建议的值，也可以拒绝它。在系统字体中 tmExternalLeading 可以为 0，因此我没有在图 4-3 中显示它。（不要怪我不告诉你们，图 4-3 是 Windows 在 640 × 480 的显示分辨率中使用的精确的系统字体。）

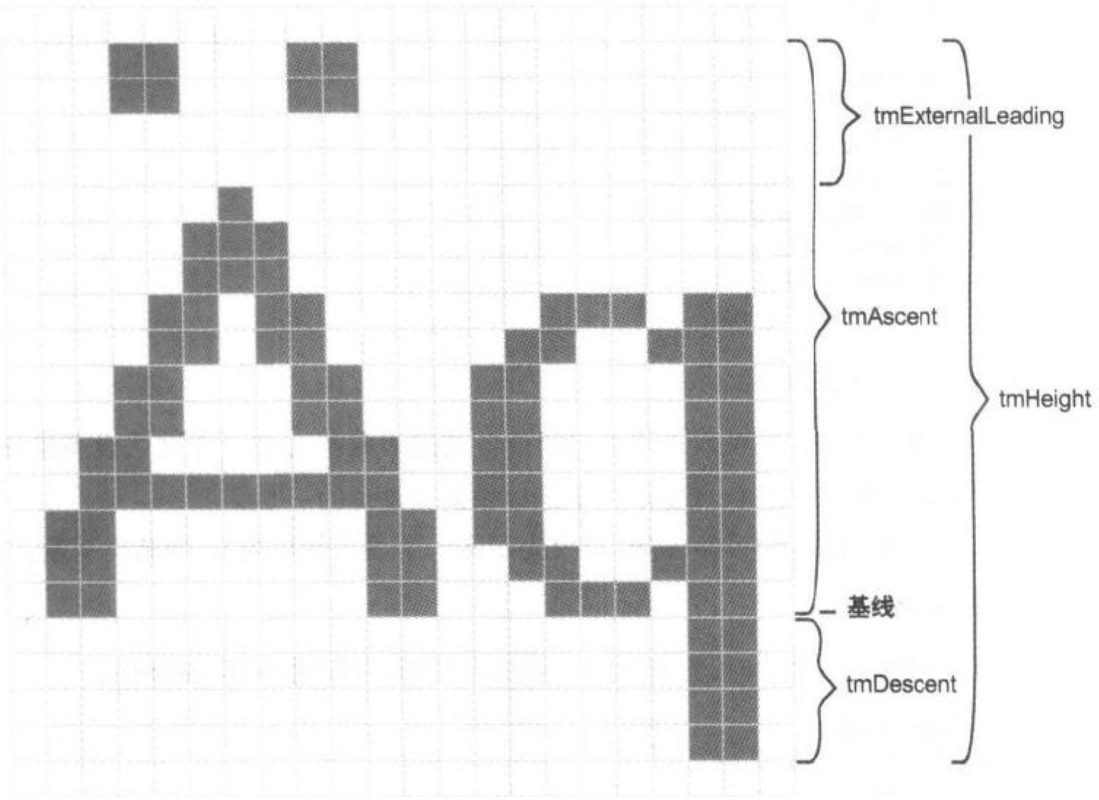


图 4-3　定义字体中纵向字符大小的 4 个值

TEXTMETRICS 结构包含有描述字符宽度的两个字段，即 tmAveCharWidth（小写字母加权平均宽度）和 tmMaxCharWidth（字体中最宽字符的宽度）。对于等宽字体，这两个值是相等的。（图 4-3 中这些值分别为 7 和 14。）

本章的示例程序还需要另一种字符宽度，即大写字母的平均宽度，这可以用 tmAveCharWidth 乘以 150% 大致计算出来。

必须认识到，系统字体的大小取决于 Windows 所运行的视频显示器的分辨率，在某些情况下，取决于用户选取的系统字体的大小。Windows 提供了一个与设备无关的图

形界面，但程序员并不是无所事事。不要想当然地猜测字符大小来编写 Windows 程序，也不要把值定死，可以使用 GetTextMetrics 函数获取这一信息。

4.2.9 格式化文本

由于在一次 Windows 对话期间，系统字体的大小不会发生改变，因此在程序运行过程中，您只需要调用一次 GetTextMetrics。最好是在窗口过程中处理 WM_CREATE 消息时进行此调用。WM_CREATE 消息是窗口过程接收的第一个消息。当在 WinMain 中调用 CreateWindow 时，Windows 以一个 WM_CREATE 消息调用窗口过程。

假设要编写一个 Windows 程序，在客户区显示几行文本，这需要先获取字符宽度和高度。您可以在窗口过程内定义两个变量来保存平均字符宽度(cxChar)和总的字符高度(cyChar)：

```
static int cxChar, cyChar ;
```

变量名的前缀 c 代表“count”，在这里指像素数，与 x 和 y 结合，分别指宽和高。这些变量定义为 static，因为它们在窗口过程中处理其他消息(如 WM_PAINT)时也应该是有效的。如果变量在函数外面定义，则不需要定义为 static。

下面是获取系统字体的字符宽度和高度的 WM_CREATE 代码：

```
case WM_CREATE:
    hdc = GetDC (hwnd) ;

    GetTextMetrics (hdc, &tm) ;
    cxChar = tm.tmAveCharWidth ;
    cyChar = tm.tmHeight + tm.tmExternalLeading ;

    ReleaseDC (hwnd, hdc) ;
    return 0 ;
```

注意我在计算 cyChar 时包括了 tmExternalLeading 字段，虽然该字段在系统字体中为 0，但是因为它使得文本的可读性更好，所以还是应该把它包括进去。沿着窗口向下每隔 cyChar 像素就会显示一行文本。

您常常会发现很有必要显示格式化的数字和简单的字符串。我在第二章讲到过，不能使用习惯的工具来完成这项工作(钟爱的 printf 函数)，但是可以使用 sprintf 和 Windows 版的 sprintf——wsprintf。这些函数与 printf 相似，只是把格式化串放到字符串中。然后，可以用 TextOut 将串输出到显示器上。非常方便的是，从 sprintf 和 wsprintf 返回的值就是串的长度。您可以将这个值传递给 TextOut 作为 iLength 参数。下面的代码显示了 wsprintf 与 TextOut 的典型组合：

```
int     iLength ;
```

```
TCHAR szBuffer [40] ;
[ other program lines]
iLength = wsprintf (szBuffer, TEXT ("The sum of %i and %i is %i"),
                    iA, iB, iA + iB) ;
TextOut (hdc, x, y, szBuffer, iLength) ;
```

对于这样简单的情况，可以将 nLength 的赋值与 TextOut 放在同一个语句中，从而无需定义 iLength：

```
TextOut (hdc, x, y, szBuffer,
        wsprintf (szBuffer, TEXT ("The sum of %i and %i is %i"),
                 iA, iB, iA + iB)) ;
```

虽然风格欠佳，但是功能与前者是一样的。

4.2.10 综合使用

现在，我们似乎已经具备了在屏幕上显示多行文本所需要的所有知识。我们知道如何在 WM_PAINT 消息期间获取一个设备描述表句柄，如何使用 TextOut 函数，以及如何根据字符大小来安排字距，剩下的就是显示一点有意义的东西了。

在前一章里，我们大概知道从 Windows 的 GetSystemMetrics 函数中获取的信息是很有意义的，该函数返回 Windows 中不同图形构件的大小信息，如图标、光标、标题栏和滚动条等。它们的大小因显示适配器和驱动程序的不同而有所不同。GetSystemMetrics 是在程序中完成与设备无关的图形输出的重要的函数。

该函数需要一个参数，叫做"索引"，索引是在 Windows 头文件中定义的 75 个整数标识符之一。（标识符的数量随着每个版本的 Windows 的发布而不断地增加，在 Windows 1.0 的程序员文档中仅列出了 26 个。）GetSystemMetrics 返回一个整数，这个整数通常是参数中指定的构件大小。

让我们来编写一个程序，显示一些可以从 GetSystemMetrics 调用中获取的信息，显示格式为每条信息占一行。如果我们建立一个头文件，在该头文件中定义一个结构数组，此结构数组包含 GetSystemMetrics 索引对应的 Windows 头文件标识符和调用所返回的每个值对应的字符串，这样处理起来要容易一些。头文件名为 SYSMETS.H，如程序 4-1 所示。

程序 4-1 SYSMETS.H 文件

SYSMETS.H

```
/*-----------------------------------------------
    SYSMETS.H -- System metrics display structure
  -----------------------------------------------*/
```

```
#define NUMLINES ((int) (sizeof sysmetrics / sizeof sysmetrics [0]))

struct
{
     int     iIndex ;
     TCHAR * szLabel ;
     TCHAR * szDesc ;
}
sysmetrics [] =
{
     SM_CXSCREEN,            TEXT ("SM_CXSCREEN "),
                             TEXT ("Screen width in pixels "),
     SM_CYSCREEN,            TEXT ("SM_CYSCREEN "),
                             TEXT ("Screen height in pixels"),
     SM_CXVSCROLL,           TEXT ("SM_CXVSCROLL "),
                             TEXT ("Vertical scroll width"),
     SM_CYHSCROLL,           TEXT ("SM_CYHSCROLL "),
                             TEXT ("Horizontal scroll height "),
     SM_CYCAPTION,           TEXT ("SM_CYCAPTION "),
                             TEXT ("Caption bar height "),
     SM_CXBORDER,            TEXT ("SM_CXBORDER "),
                             TEXT ("Window border width "),
     SM_CYBORDER,            TEXT ("SM_CYBORDER "),
                             TEXT ("Window border height "),
     SM_CXFIXEDFRAME,        TEXT ("SM_CXFIXEDFRAME "),
                             TEXT ("Dialog window frame width "),
     SM_CYFIXEDFRAME,        TEXT ("SM_CYFIXEDFRAME "),
                             TEXT ("Dialog window frame height "),
     SM_CYVTHUMB,            TEXT ("SM_CYVTHUMB "),
                             TEXT ("Vertical scroll thumb height "),
     SM_CXHTHUMB,            TEXT ("SM_CXHTHUMB "),
                             TEXT ("Horizontal scroll thumb width "),
     SM_CXICON,              TEXT ("SM_CXICON "),
                             TEXT ("Icon width "),
     SM_CYICON,              TEXT ("SM_CYICON "),
                             TEXT ("Icon height "),
     SM_CXCURSOR,            TEXT ("SM_CXCURSOR "),
                             TEXT ("Cursor width "),
     SM_CYCURSOR,            TEXT ("SM_CYCURSOR "),
                             TEXT ("Cursor height "),
     SM_CYMENU,              TEXT ("SM_CYMENU "),
                             TEXT ("Menu bar height "),
     SM_CXFULLSCREEN,        TEXT ("SM_CXFULLSCREEN "),
                             TEXT ("Full screen client area width "),
     SM_CYFULLSCREEN,        TEXT ("SM_CYFULLSCREEN "),
                             TEXT ("Full screen client area height "),
```

```
SM_CYKANJIWINDOW,      TEXT ("SM_CYKANJIWINDOW"),
                       TEXT ("Kanji window height"),
SM_MOUSEPRESENT,       TEXT ("SM_MOUSEPRESENT"),
                       TEXT ("Mouse present flag"),
SM_CYVSCROLL,          TEXT ("SM_CYVSCROLL"),
                       TEXT ("Vertical scroll arrow height"),
SM_CXHSCROLL,          TEXT ("SM_CXHSCROLL"),
                       TEXT ("Horizontal scroll arrow width"),
SM_DEBUG,              TEXT ("SM_DEBUG"),
                       TEXT ("Debug version flag"),
SM_SWAPBUTTON,         TEXT ("SM_SWAPBUTTON"),
                       TEXT ("Mouse buttons swapped flag"),
SM_CXMIN,              TEXT ("SM_CXMIN"),
                       TEXT ("Minimum window width"),
SM_CYMIN,              TEXT ("SM_CYMIN"),
                       TEXT ("Minimum window height"),
SM_CXSIZE,             TEXT ("SM_CXSIZE"),
                       TEXT ("Min/Max/Close button width"),
SM_CYSIZE,             TEXT ("SM_CYSIZE"),
                       TEXT ("Min/Max/Close button height"),
SM_CXSIZEFRAME,        TEXT ("SM_CXSIZEFRAME"),
                       TEXT ("Window sizing frame width"),
SM_CYSIZEFRAME,        TEXT ("SM_CYSIZEFRAME"),
                       TEXT ("Window sizing frame height"),
SM_CXMINTRACK,         TEXT ("SM_CXMINTRACK"),
                       TEXT ("Minimum window tracking width"),
SM_CYMINTRACK,         TEXT ("SM_CYMINTRACK"),
                       TEXT ("Minimum window tracking height"),
SM_CXDOUBLECLK,        TEXT ("SM_CXDOUBLECLK"),
                       TEXT ("Double click x tolerance"),
SM_CYDOUBLECLK,        TEXT ("SM_CYDOUBLECLK"),
                       TEXT ("Double click y tolerance"),
SM_CXICONSPACING,      TEXT ("SM_CXICONSPACING"),
                       TEXT ("Horizontal icon spacing"),
SM_CYICONSPACING,      TEXT ("SM_CYICONSPACING"),
                       TEXT ("Vertical icon spacing"),
SM_MENUDROPALIGNMENT,  TEXT ("SM_MENUDROPALIGNMENT"),
                       TEXT ("Left or right menu drop"),
SM_PENWINDOWS,         TEXT ("SM_PENWINDOWS"),
                       TEXT ("Pen extensions installed"),
SM_DBCSENABLED,        TEXT ("SM_DBCSENABLED"),
                       TEXT ("Double-Byte Char Set enabled"),
SM_CMOUSEBUTTONS,      TEXT ("SM_CMOUSEBUTTONS"),
                       TEXT ("Number of mouse buttons"),
SM_SECURE,             TEXT ("SM_SECURE"),
                       TEXT ("Security present flag"),
```

```
SM_CXEDGE,            TEXT ("SM_CXEDGE"),
                      TEXT ("3-D border width"),
SM_CYEDGE,            TEXT ("SM_CYEDGE"),
                      TEXT ("3-D border height"),
SM_CXMINSPACING,      TEXT ("SM_CXMINSPACING"),
                      TEXT ("Minimized window spacing width"),
SM_CYMINSPACING,      TEXT ("SM_CYMINSPACING"),
                      TEXT ("Minimized window spacing height"),
SM_CXSMICON,          TEXT ("SM_CXSMICON"),
                      TEXT ("Small icon width"),
SM_CYSMICON,          TEXT ("SM_CYSMICON"),
                      TEXT ("Small icon height"),
SM_CYSMCAPTION,       TEXT ("SM_CYSMCAPTION"),
                      TEXT ("Small caption height"),
SM_CXSMSIZE,          TEXT ("SM_CXSMSIZE"),
                      TEXT ("Small caption button width"),
SM_CYSMSIZE,          TEXT ("SM_CYSMSIZE"),
                      TEXT ("Small caption button height"),
SM_CXMENUSIZE,        TEXT ("SM_CXMENUSIZE"),
                      TEXT ("Menu bar button width"),
SM_CYMENUSIZE,        TEXT ("SM_CYMENUSIZE"),
                      TEXT ("Menu bar button height"),
SM_ARRANGE,           TEXT ("SM_ARRANGE"),
                      TEXT ("How minimized windows arranged"),
SM_CXMINIMIZED,       TEXT ("SM_CXMINIMIZED"),
                      TEXT ("Minimized window width"),
SM_CYMINIMIZED,       TEXT ("SM_CYMINIMIZED"),
                      TEXT ("Minimized window height"),
SM_CXMAXTRACK,        TEXT ("SM_CXMAXTRACK"),
                      TEXT ("Maximum draggable width"),
SM_CYMAXTRACK,        TEXT ("SM_CYMAXTRACK"),
                      TEXT ("Maximum draggable height"),
SM_CXMAXIMIZED,       TEXT ("SM_CXMAXIMIZED"),
                      TEXT ("Width of maximized window"),
SM_CYMAXIMIZED,       TEXT ("SM_CYMAXIMIZED"),
                      TEXT ("Height of maximized window"),
SM_NETWORK,           TEXT ("SM_NETWORK"),
                      TEXT ("Network present flag"),
SM_CLEANBOOT,         TEXT ("SM_CLEANBOOT"),
                      TEXT ("How system was booted"),
SM_CXDRAG,            TEXT ("SM_CXDRAG"),
                      TEXT ("Avoid drag x tolerance"),
SM_CYDRAG,            TEXT ("SM_CYDRAG"),
                      TEXT ("Avoid drag y tolerance"),
SM_SHOWSOUNDS,        TEXT ("SM_SHOWSOUNDS"),
                      TEXT ("Present sounds visually"),
```

```
     SM_CXMENUCHECK,            TEXT ("SM_CXMENUCHECK"),
                                TEXT ("Menu check-mark width"),
     SM_CYMENUCHECK,            TEXT ("SM_CYMENUCHECK"),
                                TEXT ("Menu check-mark height"),
     SM_SLOWMACHINE,            TEXT ("SM_SLOWMACHINE"),
                                TEXT ("Slow processor flag"),
     SM_MIDEASTENABLED,         TEXT ("SM_MIDEASTENABLED"),
                                TEXT ("Hebrew and Arabic enabled flag"),
     SM_MOUSEWHEELPRESENT,      TEXT ("SM_MOUSEWHEELPRESENT"),
                                TEXT ("Mouse wheel present flag"),
     SM_XVIRTUALSCREEN,         TEXT ("SM_XVIRTUALSCREEN"),
                                TEXT ("Virtual screen x origin"),
     SM_YVIRTUALSCREEN,         TEXT ("SM_YVIRTUALSCREEN"),
                                TEXT ("Virtual screen y origin"),
     SM_CXVIRTUALSCREEN,        TEXT ("SM_CXVIRTUALSCREEN"),
                                TEXT ("Virtual screen width"),
     SM_CYVIRTUALSCREEN,        TEXT ("SM_CYVIRTUALSCREEN"),
                                TEXT ("Virtual screen height"),
     SM_CMONITORS,              TEXT ("SM_CMONITORS"),
                                TEXT ("Number of monitors"),
     SM_SAMEDISPLAYFORMAT,      TEXT ("SM_SAMEDISPLAYFORMAT"),
                                TEXT ("Same color format flag")
} ;
```

显示信息的程序命名为 SYSMETS1。SYSMETS1.C 的源代码如程序 4-2 所示。现在大多数代码看起来都很熟悉。WinMain 中的代码实际上与 HELLOWIN 中的代码相同，并且 WndProc 中的大部分代码已经讨论过了。

程序 4-2　SYSMETS1.C

SYSMETS1.C

```
/*---------------------------------------------------
     SYSMETS1.C——System Metrics Display Program No. 1
                 (c) Charles Petzold, 1998
  ---------------------------------------------------*/

#include <windows.h>
#include "sysmets.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("SysMets1") ;
```

```
     HWND      hwnd ;
     MSG       msg ;
     WNDCLASS  wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Get System Metrics No. 1"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;
     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;
     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static int   cxChar, cxCaps, cyChar ;
     HDC          hdc ;
     int          i ;
     PAINTSTRUCT  ps ;
     TCHAR        szBuffer [10] ;
     TEXTMETRIC   tm ;

     switch (message)
```

```
    }
    case WM_CREATE:
         hdc = GetDC (hwnd) ;

         GetTextMetrics (hdc, &tm) ;
         cxChar = tm.tmAveCharWidth ;
         cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
         cyChar = tm.tmHeight + tm.tmExternalLeading ;

         ReleaseDC (hwnd, hdc) ;
         return 0 ;
    case WM_PAINT :
         hdc = BeginPaint (hwnd, &ps) ;

         for (i = 0 ; i < NUMLINES ; i++ )
         {
              TextOut (hdc, 0, cyChar * i,
                       sysmetrics[i].szLabel,
                       lstrlen (sysmetrics[i].szLabel)) ;

              TextOut (hdc, 22 * cxCaps, cyChar * i,
                       sysmetrics[i].szDesc,
                       lstrlen (sysmetrics[i].szDesc)) ;

              SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;

              TextOut (hdc, 22 * cxCaps + 40 * cxChar, cyChar * i, szBuffer,
                       wsprintf (szBuffer, TEXT ("%5d"),
                                 GetSystemMetrics (sysmetrics[i].iIndex))) ;
              SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
         }
         EndPaint (hwnd, &ps) ;
         return 0 ;

    case WM_DESTROY :
         PostQuitMessage (0) ;
         return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

图 4-4 显示了在标准 VGA 上运行的 SYSMETS1。在程序客户区的前两行可以看到，屏幕宽度是 640 个像素点，屏幕高度是 480 个像素点，这两个值以及程序所显示的其他值可能会因视频显示器类型的不同而不同。

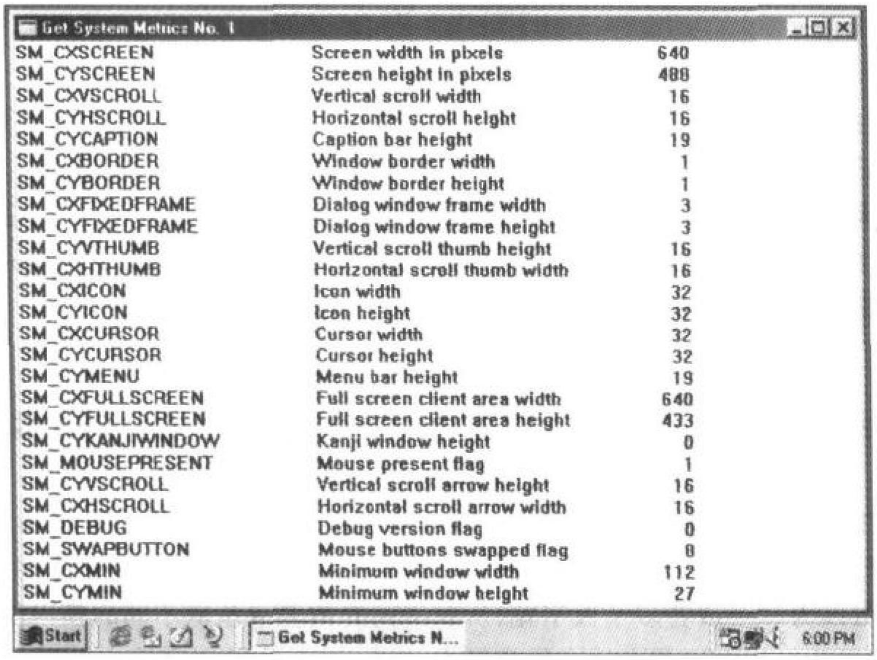


图 4-4 SYSMETS1 的显示

4.2.11 SYSMETS1.C 窗口过程

SYSMETS1.C 程序中的 WndProc 窗口过程处理三个消息：WM_CREATE、WM_PAINT 和 WM_DESTROY。WM_DESTROY 消息的处理方法与第三章的 HELLOWIN 程序相同。

WM_CREATE 消息是窗口过程接收到的第一个消息。在 CreateWindow 函数创建窗口时，Windows 产生这个消息。在处理 WM_CREATE 消息时，SYSMETS1 调用 GetDC 获取窗口的设备描述表，并调用 GetTextMetrics 获取默认系统字体的文本大小。SYSMETS1 将平均字符宽度保存在 cxChar 中，将字符的总高度（包括外部间距）保存在 cyChar 中。

SYSMETS1 还将大写字母的平均宽度保存在静态变量 cxCaps 中。对于等宽字体，cxCaps 等于 cxChar。对于变宽字体，cxCaps 设置为 cxChar 乘以 150%。对于变宽字体，TEXTMETRIC 结构中的 tmPitchAndFamily 域的低位为 1，对于等宽字体，该值为 0。SYSMETS1 使用这个位从 cxChar 计算 cxCaps：

```
cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
```

SYSMETS1 在处理 WM_PAINT 消息期间完成所有窗口创建工作。通常，窗口过程先调用 BeginPaint 获取设备描述表句柄，然后用一个 for 语句对 SYSMETS.H 中定义的 sysmetrics 结构的所有行进行循环。三列文本通过三个 TextOut 函数调用显示，对于每一列，TextOut 的第三个参数都设置为：

```
cyChar * i
```

这个参数指示了字符串顶部相对于客户区顶部的像素点位置。

第一个 TextOut 语句在第一列显示了大写标识符。TextOut 的第二个参数是 0，这是说文本从客户区的左边缘开始。文本的内容来自 sysmetrics 结构的 szLabel 字段。我使用 Windows 函数 lstrlen 来计算字符串的长度，它是 TextOut 需要的最后一个参数。

第二条 TextOut 语句显示了对系统尺寸值的描述。这些描述存放在 sysmetrics 结构的 szDesc 字段中。在这种情况下，TextOut 的第二个参数设置为：

```
22 * cxCaps
```

第一列显示的最长的大写标识符有 20 个字符，因此第二列必须在第一列文本开头向右 20 × cxCaps 处开始。我使用 22，以在两列之间加一点多余的空间。

第三条 TextOut 语句显示从 GetSystemMetrics 函数获取的数值。变宽字体使得格式化右对齐的数值有些棘手。从 0 到 9 的数字具有相同的宽度，但是这个宽度比空格宽度大。数值可以为一个或者多个数字宽，所以不同的数值应该从不同的横向位置开始。

那么，如果我们指定字符串结束的像素点位置，而不是指定字符串的开始位置，以此来右对齐数值，是否会容易一些呢？用 SetTextAlign 函数就可以做到这一点。在 SYSMETS1 调用

```
SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;
```

之后，传给后续 TextOut 函数的坐标将指定文本串的右上角，而不是左上角。

显示列数的 TextOut 函数的第二个参数设置为：

```
22 * cxCaps + 40 * cxChar
```

值 40 × cxChar 包含了第二列的宽度和第三列的宽度。在 TextOut 函数之后，另一个对 SetTextAlign 的调用将对齐方式设置回普通方式，以进行下次循环。

4.2.12 空间不够

在 SYSMETS1 程序中存在着一个很难处理的问题：除非您具有一个大屏幕、高分辨率的视频适配器，否则就无法看到系统尺度列表的最后几行。如果窗口太窄，甚至根本看不到值。

SYSMETS1 不知道这个问题。否则我们就会显示一个消息框说“Sorry！”程序甚至不知道它的客户区有多大，它从窗口顶部开始输出文本，并依靠 Windows 裁剪超出客户区底部的内容。

显然，这很不理想。为了解决这个问题，我们的第一个任务是确定程序在客户区内能输出多少内容。

4.2.13 客户区的大小

如果您使用过现有的 Windows 应用程序，可能会发现窗口的尺寸变化极大。窗口最大化时（假定窗口只有标题栏并且没有菜单），客户区几乎占据了整个屏幕。这一最大化了的客户区的尺寸可以通过以 SM_CXFULLSCREEN 和 SM_CYFULLSCREEN 为参数调用 GetSystemMetrics 来获得。窗口的最小尺寸可以很小，有时甚至不存在，更不用说客户区了。

在最近一章，我们使用 GetClientRect 函数来确定客户区的大小。使用这个函数没有什么错，但是在您每次使用信息时调用它是没有效率的。确定窗口客户区大小的更好方法是在窗口过程中处理 WM_SIZE 消息。在窗口大小改变时，Windows 给窗口过程发送一个 WM_SIZE 消息。传给窗口过程的 lParam 参数的低位字中包含客户区的宽度，高位字中包含客户区的高度。要保存这些尺寸，需要在窗口过程中定义两个静态变量：

```
static int cxClient, cyClient ;
```

与 cxChar 和 cyChar 相似，这两个变量在窗口过程内定义为静态变量，因为在以后处理其他消息时会用到它们。处理 WM_SIZE 的方法如下：

```
case WM_SIZE:
    cxClient = LOWORD (lParam) ;
    cyClient = HIWORD (lParam) ;
    return 0 ;
```

实际上您会在每个 Windows 程序中看到类似的代码。LOWORD 和 HIWORD 宏在 Windows 头文件 WINDEF.H 中定义。这些宏的定义看起来像这样：

```
#define LOWORD(l) ((WORD)(l))
#define HIWORD(l) ((WORD)(((DWORD)(l) >> 16) & 0xFFFF))
```

这两个宏返回 WORD 值（16 位的无符号整数，范围从 0 到 0xFFFF）。一般，将这些值保存在 32 位带符号整数中。这就不会牵扯到任何转换问题，并使得这些值在以后需要的任何计算中易于使用。

在许多 Windows 程序中，WM_SIZE 消息必然跟着一个 WM_PAINT 消息。为什么呢？因为在我们定义窗口类时指定类风格为

```
CS_HREDRAW | CS_VREDRAW
```

这种类风格告诉 Windows，如果水平或者垂直大小发生改变，则强行刷新客户区。

用如下公式计算可以在客户区内显示的文本的总行数：

```
cyClient / cyChar
```

如果客户区的高度太小以至于无法显示一个完整的字符，这个公式的结果可以为0。类似地，在客户区的水平方向可以显示的小写字符的近似数目为

```
cxClient / cxChar
```

如果在处理 WM_CREATE 消息期间确定 cxChar 和 cyChar，则不用担心在这两个计算公式中会出现被0除的情况。在 WinMain 调用 CreateWindow 时，窗口过程接收一个 WM_CREATE 消息。在 WinMain 调用 ShowWindow 之后接收到第一个 WM_CREATE 消息，此时 cxChar 和 cyChar 已经被赋予正的非零值了。

如果客户区的大小不足以容纳所有的内容，那么，知道窗口客户区的大小只是为用户提供了在客户区内移动文本的第一步。如果您对其他有类似需求的 Windows 应用程序很熟悉，就很可能知道，这种情况下，我们需要使用“滚动条”。

4.3 滚动条

滚动条是图形用户界面中最好的功能之一，它很容易使用，而且提供了很好的视觉反馈。您可以使用滚动条显示任何东西——无论是文本、图形、表格、数据库记录、图像，还是 Web 页面，只要它所需的空间超出了窗口的客户区所能提供的空间，就可以使用滚动条。

滚动条既有垂直方向的(供上下移动)，也有水平方向的(供左右移动)。用户可以使用鼠标在滚动条两端的箭头上或者在箭头之间的区域中点一下，这时，“滚动框”在滚动条内的移动模拟了所显示的信息在整个文档中的近似相关位置。用户也可以用鼠标拖动滚动框到特定的位置。下页图4-5显示了推荐使用的垂直滚动条。

有时，程序员对滚动概念很难理解，因为他们的观点与用户的观点不同：用户向下滚动是想看到文档稍下的部分；但是，程序实际上是将文档相对于显示窗口向上移动。Windows 文档和头文件标识符是基于用户的观点：向上滚动意味着朝文档的开头移动；向下滚动意味着朝文档尾部移动。

很容易在应用程序中包含水平或者垂直的滚动条，程序员只需要在 CreateWindow 的第三个参数中包括窗口风格(WS)标识符 WS_VSCROLL(垂直滚动)和(或)WS_HSCROLL(水平滚动)即可。这些滚动条通常放在窗口的右部和底部，伸展为客户区的整个长度或宽度。客户区不包含滚动条所占据的空间。对于特定的显示驱动程序和显示分辨率，垂直滚动条的宽度和水平滚动条的高度是恒定的。如果需要这些值，可以使用 GetSystemMetrics 调用来获取(如前面的程序那样)。

Windows 负责处理对滚动条的所有鼠标操作，但是，窗口滚动条没有自动的键盘界面。如果想用光标键来完成滚动功能，则必须提供这方面的代码(我们将在下一章另一

个版本的 SYSMETS 程序中做到这一点)。

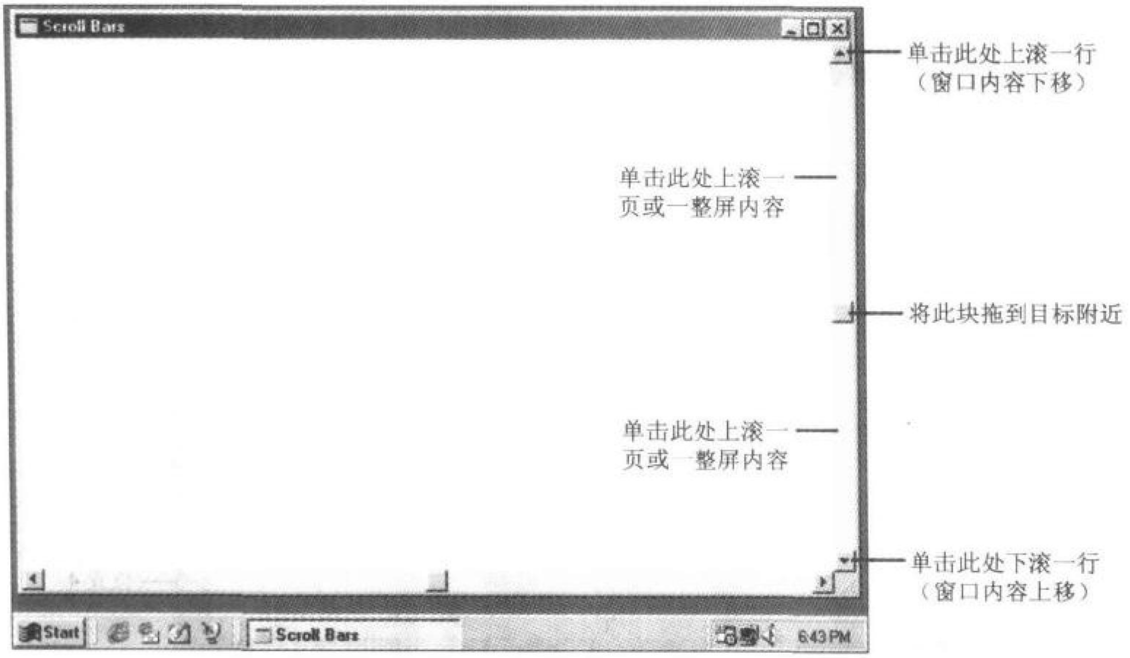


图 4-5 垂直滚动条

4.3.1 滚动条的范围和位置

每个滚动条均有一个相关的"范围"(这是一对整数,分别代表最小值和最大值)和"位置"(它是滚动框在此范围内的位置)。当滚动框在滚动条的顶部(或左部)时,滚动框的位置是范围的最小值;在滚动条的底部(或右部)时,滚动框的位置是范围的最大值。

在默认情况下,滚动条的范围是从 0(顶部或左部)~100(底部或右部),但将范围改变为更方便于程序的数值也是很容易的:

```
SetScrollRange (hwnd, iBar, iMin, iMax, bRedraw) ;
```

参数 iBar 为 SB_VERT 或者 SB_HORZ,iMin 和 iMax 分别是范围的最小值和最大值。如果想要 Windows 根据新范围重画滚动条,则设置 bRedraw 为 TRUE。(如果在调用 SetScrollRange 后,调用了影响滚动条位置的其他函数,则应该将 bRedraw 设置为 FALSE,以避免过多的重画。)

滚动框的位置总是离散的整数值。例如,范围为 0~4 的滚动条具有 5 个滚动框位置,如图 4-6 所示。

您可以使用 SetScrollPos 在滚动条范围内设置新的滚动框位置:

```
SetScrollPos (hwnd, iBar, iPos, bRedraw) ;
```

参数 iPos 是新位置，它必须在 iMin 至 iMax 的范围内。Windows 提供了类似的函数(GetScrollRange 和 GetScrollPos)来获取滚动条的当前范围和位置。

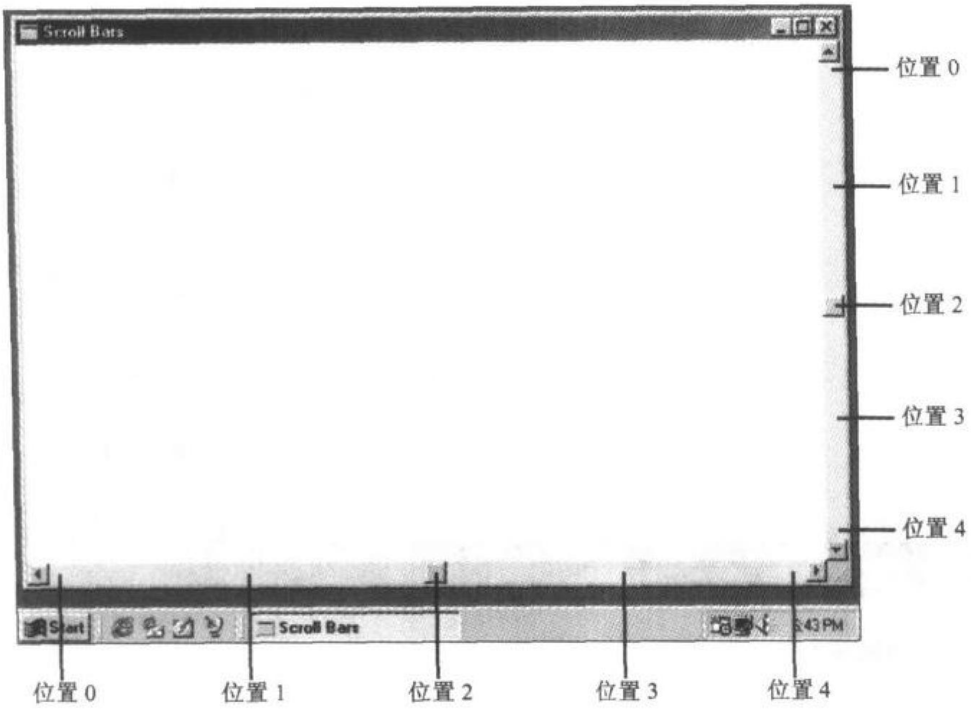


图 4-6 具有 5 个滚动框位置的滚动条

在程序内使用滚动条时，程序员与 Windows 共同负责维护滚动条以及更新滚动框的位置。下面是 Windows 对滚动条的处理：

- 处理所有滚动条鼠标事件；
- 当用户在滚动条内单击鼠标时，提供一种“反相视频”闪烁；
- 当用户在滚动条内拖动滚动框时，移动滚动框；
- 为包含滚动条窗口的窗口过程发送滚动条消息。

以下是程序员应该完成的工作：

- 初始化滚动条的范围和位置；
- 处理窗口过程的滚动条消息；
- 更新滚动条内滚动框的位置；
- 更改客户区的内容以响应对滚动条的更改。

像生活中的大多数事情一样，在我们看一些代码时这些会显得更加有意义。

4.3.2 滚动条消息

在用鼠标单击滚动条或者拖动滚动框时，Windows 给窗口过程发送 WM_VSCROLL(供上下移动)和 WM_HSCROLL(供左右移动)消息。在滚动条上的每个鼠标动作都至少产生两个消息，一个在按下鼠标键时产生，一个在释放键时产生。

与所有的消息一样，WM_VSCROLL 和 WM_HSCROLL 也带有 wParam 和 lParam 消息参数。对于来自作为窗口的一部分而创建的滚动条消息，您可以忽略 lParam；它只用于作为子窗口而创建的滚动条(通常在对话框内)。

wParam 消息参数被分为一个低位字和一个高位字。其的低位字是一个数值，它指出了鼠标对滚动条进行的操作。这个数值被看作一个"通知码"。通知码的值由以 SB (代表"scroll bar(滚动条)")开头的标识符定义。以下是在 WINUSER.H 中定义的通知码：

```
#define SB_LINEUP           0
#define SB_LINELEFT         0
#define SB_LINEDOWN         1
#define SB_LINERIGHT        1
#define SB_PAGEUP           2
#define SB_PAGELEFT         2
#define SB_PAGEDOWN         3
#define SB_PAGERIGHT        3
#define SB_THUMBPOSITION    4
#define SB_THUMBTRACK       5
#define SB_TOP              6
#define SB_LEFT             6
#define SB_BOTTOM           7
#define SB_RIGHT            7
#define SB_ENDSCROLL        8
```

包含 LEFT 和 RIGHT 的标识符用于水平滚动条，包含 UP、DOWN、TOP 和 BOTTOM 的标识符用于垂直滚动条。鼠标在滚动条的不同区域单击所产生的通知码如下页图 4-7 所示。

如果在滚动条的各个部位按住鼠标键，程序就能收到多个滚动条消息。当释放鼠标键后，程序会收到一个带有 SB_ENDSCROLL 通知码的消息。一般可以忽略这个消息，Windows 不会去改变滚动框的位置，而您可以在程序中调用 SetScrollPos 来改变滚动框的位置。

当把鼠标的光标放在滚动框上并按住鼠标键时，您就可以移动滚动框。这样就产生了带有 SB_THUMBTRACK 和 SB_THUMBPOSITION 通知码的滚动条消息。在 wParam

的低位字是 SB_THUMBTRACK 时,wParam 的高位字是用户在拖动滚动框时的当前位置。该位置位于滚动条范围的最小值和最大值之间。在 wParam 的低位字是 SB_THUMBPOSITION 时,wParam 的高位字是用户释放鼠标键后滚动框的最终位置。对于其他的滚动条操作,wParam 的高位字应该被忽略。

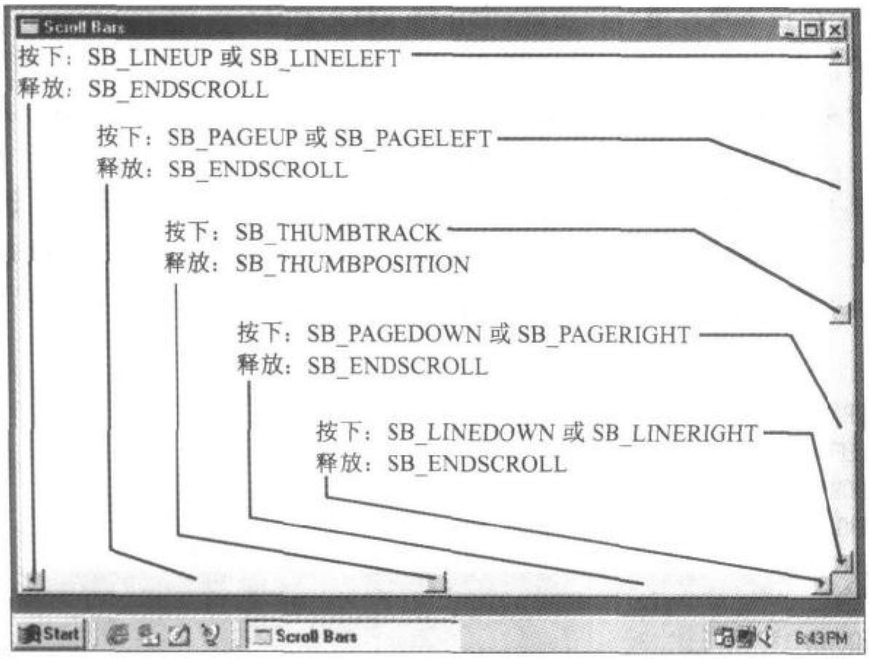


图 4-7 滚动条消息的 wParam 值所对应的标识符

为了给用户提供反馈,Windows 在您用鼠标拖动滚动框时移动它,同时您的程序会收到 SB_THUMBTRACK 消息。然而,如果不通过调用 SetScrollPos 来处理 SB_THUMBTRACK 或 SB_THUMBPOSITION 消息,在用户释放鼠标键后,滚动框会迅速跳回原来的位置。

程序能够处理 SB_THUMBTRACK 或 SB_THUMBPOSITION 消息,但一般不同时处理两者。如果处理 SB_THUMBTRACK 消息,在用户拖动滚动框时您需要移动客户区的内容。而如果处理 SB_THUMBPOSITION 消息,则只需在用户停止拖动滚动框时移动客户区的内容。处理 SB_THUMBTRACK 消息更好一些(但更困难),对于某些类型的数据,您的程序可能很难跟上产生的消息。

WINUSER.H 头文件还包括 SB_TOP、SB_BOTTOM、SB_LEFT 和 SB_RIGHT 通知码,指出滚动条已经被移到了它的最小或最大位置。然而,对于作为应用程序窗口一部分而创建的滚动条来说,永远不会接收到这些通知码。

为滚动条范围使用 32 位的值也是有效的,尽管这不常见。然而,wParam 的高位字

只是 16 位的,它不能适当地指出 SB_THUMBTRACK 和 SB_THUMBPOSITION 操作的位置。在这种情况下,需要使用 GetScrollInfo 函数(在下面描述)来得到信息。

4.3.3 在 SYSMETS 中添加滚动功能

前面的说明已经很详尽了,现在,要将其应用于实践中。让我们开始时简单些,从垂直滚动入手,因为我们实在太需要垂直滚动了,而暂时还可以不用水平滚动。SYSMET2 如程序 4-3 所示。这个程序可能是滚动条的最简单的应用。

程序 4-3 SYSMETS2 程序

SYSMETS2.C

```
/*-----------------------------------------------------
   SYSMETS2.C——System Metrics Display Program No. 2
               (c) Charles Petzold, 1998
  -----------------------------------------------------*/

#include <windows.h>
#include "sysmets.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("SysMets2") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
```

```
                    szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Get System Metrics No. 2"),
                          WS_OVERLAPPEDWINDOW | WS_VSCROLL,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;
     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}
LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static int cxChar, cxCaps, cyChar, cyClient, iVscrollPos ;
     HDC          hdc ;
     int          i, y ;
     PAINTSTRUCT  ps ;
     TCHAR        szBuffer[10] ;
     TEXTMETRIC   tm ;

     switch (message)
     {
     case WM_CREATE:
          hdc = GetDC (hwnd) ;

          GetTextMetrics (hdc, &tm) ;
          cxChar = tm.tmAveCharWidth ;
          cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
          cyChar = tm.tmHeight + tm.tmExternalLeading ;

          ReleaseDC (hwnd, hdc) ;

          SetScrollRange (hwnd, SB_VERT, 0, NUMLINES - 1, FALSE) ;
          SetScrollPos   (hwnd, SB_VERT, iVscrollPos, TRUE) ;
          return 0 ;

     case WM_SIZE:
          cyClient = HIWORD (lParam) ;
          return 0 ;
```

```
case WM_VSCROLL:
     switch (LOWORD (wParam))
     {
     case SB_LINEUP:
          iVscrollPos -= 1 ;
          break ;

     case SB_LINEDOWN:
          iVscrollPos += 1 ;
          break ;

     case SB_PAGEUP:
          iVscrollPos -= cyClient / cyChar ;
          break ;

     case SB_PAGEDOWN:
          iVscrollPos += cyClient / cyChar ;
          break ;

     case SB_THUMBPOSITION:
          iVscrollPos = HIWORD (wParam) ;
          break ;

     default :
          break ;
     }

     iVscrollPos = max (0, min (iVscrollPos, NUMLINES - 1)) ;

     if (iVscrollPos != GetScrollPos (hwnd, SB_VERT))
     {
          SetScrollPos (hwnd, SB_VERT, iVscrollPos, TRUE) ;
          InvalidateRect (hwnd, NULL, TRUE) ;
     }
     return 0 ;

case WM_PAINT:
     hdc = BeginPaint (hwnd, &ps) ;

     for (i = 0 ; i < NUMLINES ; i++ )
     {
          y = cyChar * (i - iVscrollPos) ;

          TextOut (hdc, 0, y,
                   sysmetrics[i].szLabel,
                   lstrlen (sysmetrics[i].szLabel)) ;
```

```
          TextOut (hdc, 22 * cxCaps, y,
                   sysmetrics[i].szDesc,
                   lstrlen (sysmetrics[i].szDesc)) ;

          SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;
          TextOut (hdc, 22 * cxCaps + 40 * cxChar, y, szBuffer,
                   wsprintf (szBuffer, TEXT ("%5d"),
                        GetSystemMetrics (sysmetrics[i].iIndex))) ;

                   SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
          }
          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY:
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

新的 CreateWindow 调用在第三个参数中包含了 WS_VSCROLL 窗口风格，从而在窗口中加入了垂直滚动条，其窗口风格为：

```
WS_OVERLAPPEDWINDOW | WS_VSCROLL
```

WndProc 窗口过程在处理 WM_CREATE 消息时增加了两条语句，以设置垂直滚动条的范围和初始位置：

```
SetScrollRange (hwnd, SB_VERT, 0, NUMLINES - 1, FALSE) ;
SetScrollPos (hwnd, SB_VERT, iVscrollPos, TRUE) ;
```

sysmetrics 结构具有 NUMLINES 行文本，所以滚动条范围被设置为 0 至 NUMLINES - 1。滚动条的每个位置对应于在客户区顶部显示的一个文本行。如果滚动框的位置为 0，则第一行会被放置在客户区的顶部。如果位置大于 0，其他行就会出现在客户区的顶部。当位置为 NUMLINES - 1 时，则最后一行文本出现在客户区的顶部。

为了有助于处理 WM_VSCROLL 消息，在窗口过程中定义了一个静态变量 iVscrollPos，这一变量是滚动条内滚动框的当前位置。对于 SB_LINEUP 和 SB_LINEDOWN，只需要将滚动框调整一个单位的位置。对于 SB_PAGEUP 和 SB_PAGEDOWN，我们想整屏地移动屏幕的内容，或者移动 cyClient /cyChar 个单位的位置。对于 SB_THUMBPOSITION，新的滚动框位置是 wParam 的高位字。SB_ENDSCROLL 和 SB_THUMBTRACK 消息被忽略。

在程序基于收到的 WM_VSCROLL 类型的消息计算出新的 iVscrollPos 值后，用 min

和 max 宏来调整 iVscrollPos，以确保它在最大值与最小值之间。程序然后将 iVscrollPos 与调用 GetScrollPos 获取的先前位置相比较，如果滚动位置发生了变化，则使用 SetScrollPos 来进行更新，并且调用 InvalidateRect 使整个窗口无效。

InvalidateRect 调用产生一个 WM_PAINT 消息。SYSMETS1 在处理 WM_PAINT 消息时，每一行的 y 坐标计算公式为

```
cyChar * i
```

在 SYSMETS2 中，计算公式为

```
cyChar * (i - iVscrollPos)
```

循环仍然显示 NUMLINES 行文本，但是对于非零值的 iVscrollPos 是负数。程序实际上在客户区以外显示这些文本行。当然，Windows 不会显示这些行，因此屏幕显得干净和漂亮。

前面说过，我们一开始不想弄得太复杂，这样的代码很浪费，效率很低。下面我们对此加以修改，但是先要考虑在 WM_VSCROLL 消息之后刷新客户区的方法。

4.3.4 绘图程序的组织

在处理完滚动条消息后，SYSMETS2 不刷新客户区，相反，它调用 InvalidateRect 使客户区失效。这导致 Windows 将一个 WM_PAINT 消息放入消息队列中。

最好能使 Windows 程序在响应 WM_PAINT 消息时完成所有的客户区绘制功能。因为程序必须在一接收到 WM_PAINT 消息时就刷新整个客户区，如果在程序的其他部分也绘制的话，将很可能使代码重复。

首先，您可能对这种拐弯抹角的方式感到厌烦。在 Windows 的早期，因为这种方式与字符模式的 PC 编程差别太大，程序员感到这种概念很难理解。并且，程序要不断地通过马上绘制某种东西来响应键盘和鼠标。这样做既方便又有效。但是在很多情况下，这完全不必要。当您掌握了在响应 WM_PAINT 消息时积累绘制客户区所需要的全部信息这样的原则之后，会对这种结果感到满意的。

如同 SYSMETS2 示范的，程序仍然需要在处理非 WM_PAINT 消息时刷新特定的显示区域，使用 InvalidateRect 就很方便，您可以用它使客户区内的特定矩形或者整个客户区失效。

只将窗口客户区标记为无效以产生 WM_PAINT 消息，对于某些应用程序来说也许不是完全令人满意的选择。在调用 InvalidateRect 之后，Windows 将 WM_PAINT 消息放入消息队列中，最后由窗口过程处理它。然而，Windows 将 WM_PAINT 消息当成低优先级消息，如果系统有许多其他的动作正在发生，那么也许会让您等待一会儿工夫。这

时，当对话框消失时，将会出现一些空白的“洞”，程序仍然等待刷新它的窗口。

如果您希望立即刷新无效区域，可以在调用 InvalidateRect 之后调用 UpdateWindow：

```
UpdateWindow (hwnd) ;
```

如果客户区的任一部分无效，则 UpdateWindow 将导致 Windows 用 WM_PAINT 消息调用窗口过程（如果整个客户区有效，则不调用窗口过程）。这一 WM_PAINT 消息不进入消息队列，直接由 Windows 调用窗口过程。窗口过程完成刷新后立即退出，Windows 将控制返回给程序中 UpdateWindow 调用之后的语句。

您可能注意到，UpdateWindow 与 WinMain 中用来产生第一个 WM_PAINT 消息的函数相同。最初创建窗口时，整个客户区变为无效，UpdateWindow 指示窗口过程绘制客户区。

4.4 建立更好的滚动

SYSMETS2 工作得很好，但它只是模仿其他程序中的滚动条，并且效率很低。很快我将展示一个新的版本来改进它的不足。也许最有趣的是这个新版本不使用目前所讨论的 4 个滚动条函数。相反，它将使用 Win32 API 中才有的新函数。

4.4.1 滚动条信息函数

滚动条文档（在/Platform SDK/User Interface Services/Controls/Scroll Bars 中）指出 SetScrollRange、SetScrollPos、GetScrollRange 和 GetScrollPos 函数是“过时的”，但这并不完全正确。这些函数在 Windows 1.0 中就出现了，在 Win32 API 中升级以处理 32 位参数。它们仍然具有良好的功能。而且，它们不与 Windows 编程中新函数相冲突，这就是我在此书中仍使用它们的原因。

Win32 API 介绍的两个滚动条函数称作 SetScrollInfo 和 GetScrollInfo。这些函数完成以前函数的全部功能，并增加了两个新特性。

第一个功能涉及滚动框的大小。您可能注意到，滚动框大小在 SYSMETS2 程序中是固定的。然而，在您可能使用到的一些 Windows 应用程序中，滚动框大小与在窗口中显示的文档大小成比例。显示的大小称作“页面大小”。算法为：

$$\frac{\text{滚动框大小}}{\text{滚动长度}} \approx \frac{\text{页面大小}}{\text{范围}} \approx \frac{\text{显示的文档数量}}{\text{文档的总大小}}$$

可以使用 SetScrollInfo 来设置页面大小（从而设置了滚动框的大小），如将要看到的 SYSMETS3 程序所示。

GetScrollInfo 函数增加了第二个重要的功能，或者说它改进了目前 API 的不足。假设您要使用 65 536 或更大单位的范围，这在 16 位 Windows 中是不可能的。当然在

Win32 中,函数被定义为可接受 32 位参数,因此是没有问题的。(记住,如果使用这样大的范围,滚动框的实际物理位置数仍然由滚动条的像素大小限制。)然而,当使用 SB_THUMBTRACK 或 SB_THUMBPOSITION 通知码得到 WM_VSCROLL 或 WM_HSCROLL 消息时,只提供了 16 位数据来指出滚动框的当前位置。通过 GetScrollInfo 函数可以获取真实的 32 位值。

SetScrollInfo 和 GetScrollInfo 函数的语法是

```
SetScrollInfo (hwnd, iBar, &si, bRedraw) ;
GetScrollInfo (hwnd, iBar, &si) ;
```

像在其他滚动条函数中那样,iBar 参数是 SB_VERT 或 SB_HORZ,它还可以是用于滚动条控制的 SB_CTL。SetScrollInfo 的最后一个参数可以是 TRUE 或 FALSE,指出了是否要 Windows 重新绘制计算了新信息后的滚动条。

两个函数的第三个参数是 SCROLLINFO 结构,定义为:

```
typedef struct tagSCROLLINFO
{
     UINT cbSize ;          // set to sizeof (SCROLLINFO)
     UINT fMask ;           // values to set or get
     int nMin ;             // minimum range value
     int nMax ;             // maximum range value
     UINT nPage ;           // page size
     int nPos ;             // current position
     int nTrackPos ;        // current tracking position
}
SCROLLINFO, * PSCROLLINFO ;
```

在程序中,可以定义如下的 SCROLLINFO 结构类型:

```
SCROLLINFO si ;
```

在调用 SetScrollInfo 或 GetScrollInfo 之前,必须将 cbSize 字段设置为结构的大小:

```
si.cbSize = sizeof (si) ;
```

或

```
si.cbSize = sizeof (SCROLLINFO) ;
```

逐渐熟悉 Windows 后,您就会发现另外几个结构像这个结构一样,第一个字段指出了结构大小。这个字段使将来的 Windows 版本可以扩充结构并添加新的功能,并且仍然与以前编译的版本兼容。

把 fMask 字段设置为以 SIF 前缀开头的一个或多个标志,并且可以使用 C 的位操作 OR 函数(|)组合这些标志。

当通过 SetScrollInfo 函数使用 SIF_RANGE 标志时,必须把 nMin 和 nMax 字段设置为

所需的滚动条范围。当通过 GetScrollInfo 函数使用 SIF_RANGE 标志时，应把 nMin 和 nMax 字段设置为从函数返回的当前范围。

SIF_POS 标志也一样。当通过 SetScrollInfo 使用它时，必须把结构的 nPos 字段设置为所需的位置。可以通过 GetScrollInfo 使用 SIF_POS 标志来获取当前位置。

使用 SIF_PAGE 标志能够获取页面大小。用 SetScrollInfo 函数把 nPage 设置为所需的页面大小。GetScrollInfo 使用 SIF_PAGE 标志可以获取当前页面的大小。如果不想得到比例化的滚动条，就不要使用该标志。

当处理带有 SB_THUMBTRACK 或 SB_THUMBPOSITION 通知码的 WM_VSCROLL 或 WM_HSCROLL 消息时，通过 GetScrollInfo 只使用 SIF_TRACKPOS 标志。从函数的返回中，SCROLLINFO 结构的 nTrackPos 字段将指出当前的 32 位滚动框位置。

在 SetScrollInfo 函数中仅使用 SIF_DISABLENOSCROLL 标志。如果指定了此标志，并且新的滚动条参数使滚动条不可见，则禁用该滚动条。（下面会有更多的解释。）

SIF_ALL 标志是 SIF_RANGE、SIF_POS、SIF_PAGE 和 SIF_TRACKPOS 的组合。当在 WM_SIZE 消息期间设置滚动条参数时，这是很方便的。（当在 SetScrollInfo 函数中指定 SIF_TRACKPOS 时，则 SIF_TRACKPOS 标志被忽略。）这在处理滚动条消息时也是很方便的。

4.4.2 滚动范围

在 SYSMETS2 中，滚动范围设置最小为 0，最大为 NUMLINES - 1。当滚动条位置是 0 时，第一行信息显示在客户区的顶部；当滚动条的位置是 NUMLINES - 1 时，最后一行显示在客户区的顶部，并且看不见其他行。

可以说 SYSMETS2 滚动范围太大。事实上只需把信息最后一行显示在客户区的底部而不是顶部即可。我们可以对 SYSMETS2 做出一些修改以达到此点。当处理 WM_CREATE 消息时不设置滚动条范围，而是等到接收到 WM_SIZE 消息后再做此工作：

```
iVscrollMax = max (0, NUMLINES - cyClient / cyChar) ;
SetScrollRange (hwnd, SB_VERT, 0, iVscrollMax, TRUE) ;
```

假定 NUMLINES 等于 75，并假定特定窗口大小是 50(cyClient 除以 cyChar)。换句话说，我们有 75 行信息但只有 50 行可以显示在客户区中。使用上面的两行代码，把范围设置最小为 0，最大为 25。当滚动条位置等于 0 时，程序显示 0 到 49 行。当滚动条位置等于 1 时，程序显示 1 到 50 行；并且当滚动条位置等于 25(最大值)时，程序显示 25 到 74 行。很明显，需要对程序的其他部分做出修改，但这是可行的。

新滚动条函数的一个好的功能是当使用与滚动条范围一样大的页面时，它已经为您做出了大量的逻辑功能。可以像下面的代码一样使用 SCROLLINFO 结构和 SetScroll-

Info：

```
si.cbSize = sizeof (SCROLLINFO) ;
si.cbMask= SIF_RANGE | SIF_PAGE ;
si.nMin  = 0 ;
si.nMax  = NUMLINES - 1 ;
si.nPage = cyClient / cyChar ;
SetScrollInfo (hwnd, SB_VERT, &si, TRUE) ;
```

当这样做之后，Windows 会把最大的滚动条位置限制为 si.nMax － si.nPage ＋1，而不是 si.nMax。像前面那样做出假设：NUMLINES 等于 75（所以 si.nMax 等于 74），si.nPage 等于 50。这意味着最大的滚动条位置限制为 74－50＋1，即 25。这正是我们想要的。

当页面大小与滚动条范围一样大时，会发生什么情况呢？在这个例子中，就是 nPage 等于 75 或更大的情况。Windows 通常隐藏滚动条，因为它并不需要。如果不想隐藏滚动条，可在调用 SetScrollInfo 时使用 SIF_DISABLENOSCROLL，Windows 只是禁用滚动条而不隐藏它。

4.4.3 新 SYSMETS

SYSMETS3——此章中最后的 SYSMETS 程序版本——显示在程序 4-4 中。此版本使用 SetScrollInfo 和 GetScrollInfo 函数，添加左右滚动的水平滚动条，并能更有效地重画客户区。

程序 4-4 SYSMETS3 程序

SYSMETS3.C

```
/*------------------------------------------------------------
     SYSMETS3.C——System Metrics Display Program No. 3
                 (c) Charles Petzold, 1998
  ------------------------------------------------------------*/

#include <windows.h>
#include "sysmets.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)

{
     static TCHAR szAppName[] = TEXT ("SysMets3") ;
     HWND         hwnd ;
     MSG          msg ;
```

```
     WNDCLASS   wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Get System Metrics No. 3"),
                          WS_OVERLAPPEDWINDOW | WS_VSCROLL | WS_HSCROLL,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;
     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static int  cxChar, cxCaps, cyChar, cxClient, cyClient, iMaxWidth ;
     HDC         hdc ;
     int         i, x, y, iVertPos, iHorzPos, iPaintBeg, iPaintEnd ;
     PAINTSTRUCT ps ;
     SCROLLINFO  si ;
     TCHAR       szBuffer[10] ;
     TEXTMETRIC  tm ;

     switch (message)
```

```
}
case WM_CREATE:
    hdc = GetDC (hwnd) ;

    GetTextMetrics (hdc, &tm) ;
    cxChar = tm.tmAveCharWidth ;
    cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
    cyChar = tm.tmHeight + tm.tmExternalLeading ;

    ReleaseDC (hwnd, hdc) ;

         // Save the width of the three columns

    iMaxWidth = 40 * cxChar + 22 * cxCaps ;
    return 0 ;

case WM_SIZE:
    cxClient = LOWORD (lParam) ;
    cyClient = HIWORD (lParam) ;

         // Set vertical scroll bar range and page size

    si.cbSize = sizeof (si) ;
    si.fMask  = SIF_RANGE | SIF_PAGE ;
    si.nMin   = 0 ;
    si.nMax   = NUMLINES - 1 ;
    si.nPage  = cyClient / cyChar ;
    SetScrollInfo (hwnd, SB_VERT, &si, TRUE) ;

         // Set horizontal scroll bar range and page size

    si.cbSize = sizeof (si) ;
    si.fMask  = SIF_RANGE | SIF_PAGE ;
    si.nMin   = 0 ;
    si.nMax   = 2 + iMaxWidth / cxChar ;
    si.nPage  = cxClient / cxChar ;
    SetScrollInfo (hwnd, SB_HORZ, &si, TRUE) ;
    return 0 ;

case WM_VSCROLL:
         // Get all the vertical scroll bar information

    si.cbSize = sizeof (si) ;
    si.fMask  = SIF_ALL ;
    GetScrollInfo (hwnd, SB_VERT, &si) ;

         // Save the position for comparison later on
```

```
iVertPos = si.nPos ;

switch (LOWORD (wParam))
{
case SB_TOP:
     si.nPos = si.nMin ;
     break ;

case SB_BOTTOM:
     si.nPos = si.nMax ;
     break ;

case SB_LINEUP:
     si.nPos -= 1 ;
     break ;

case SB_LINEDOWN:
     si.nPos += 1 ;
     break ;

case SB_PAGEUP:
     si.nPos -= si.nPage ;
     break ;

case SB_PAGEDOWN:
     si.nPos += si.nPage ;
     break ;

case SB_THUMBTRACK:
     si.nPos = si.nTrackPos ;
     break ;

default:
     break ;
}
        // Set the position and then retrieve it. Due to adjustments
        // by Windows it may not be the same as the value set.

si.fMask = SIF_POS ;
SetScrollInfo (hwnd, SB_VERT, &si, TRUE) ;
GetScrollInfo (hwnd, SB_VERT, &si) ;

        // If the position has changed, scroll the window and update it

if (si.nPos != iVertPos)
{
     ScrollWindow (hwnd, 0, cyChar * (iVertPos - si.nPos),
```

```
                    NULL, NULL) ;
          UpdateWindow (hwnd) ;
     }
     return 0 ;

case WM_HSCROLL:
               // Get all the vertical scroll bar information

     si.cbSize = sizeof (si) ;
     si.fMask  = SIF_ALL ;

               // Save the position for comparison later on

     GetScrollInfo (hwnd, SB_HORZ, &si) ;
     iHorzPos = si.nPos ;

     switch (LOWORD (wParam))
     {
     case SB_LINELEFT:
          si.nPos -= 1 ;
          break ;

     case SB_LINERIGHT:
          si.nPos += 1 ;
          break ;

     case SB_PAGELEFT:
          si.nPos -= si.nPage ;
          break ;

     case SB_PAGERIGHT:
          si.nPos += si.nPage ;
          break ;

     case SB_THUMBPOSITION:
          si.nPos = si.nTrackPos ;
          break ;

     default :
          break ;
     }
               // Set the position and then retrieve it. Due to adjustments
               // by Windows it may not be the same as the value set.
     si.fMask = SIF_POS ;
     SetScrollInfo (hwnd, SB_HORZ, &si, TRUE) ;
     GetScrollInfo (hwnd, SB_HORZ, &si) ;
```

```
               // If the position has changed, scroll the window

          if (si.nPos != iHorzPos)
          {
               ScrollWindow (hwnd, cxChar * (iHorzPos - si.nPos), 0,
                             NULL, NULL) ;
          }
          return 0 ;

     case WM_PAINT :
          hdc = BeginPaint (hwnd, &ps) ;

               // Get vertical scroll bar position

          si.cbSize = sizeof (si) ;
          si.fMask  = SIF_POS ;
          GetScrollInfo (hwnd, SB_VERT, &si) ;
          iVertPos = si.nPos ;

               // Get horizontal scroll bar position

          GetScrollInfo (hwnd, SB_HORZ, &si) ;
          iHorzPos = si.nPos ;

               // Find painting limits

          iPaintBeg = max (0, iVertPos + ps.rcPaint.top / cyChar) ;
          iPaintEnd = min (NUMLINES - 1,
                           iVertPos + ps.rcPaint.bottom / cyChar) ;

          for (i = iPaintBeg ; i <= iPaintEnd ; i++)
          {
               x = cxChar * (1 - iHorzPos) ;
               y = cyChar * (i - iVertPos) ;

               TextOut (hdc, x, y,
                        sysmetrics[i].szLabel,
                        lstrlen (sysmetrics[i].szLabel)) ;

               TextOut (hdc, x + 22 * cxCaps, y,
                        sysmetrics[i].szDesc,
                        lstrlen (sysmetrics[i].szDesc)) ;

               SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;

               TextOut (hdc, x + 22 * cxCaps + 40 * cxChar, y, szBuffer,
                        wsprintf (szBuffer, TEXT ("%5d"),
                             GetSystemMetrics (sysmetrics[i].iIndex))) ;
```

```
            SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
        }

        EndPaint (hwnd, &ps) ;
        return 0 ;

    case WM_DESTROY :
        PostQuitMessage (0) ;
        return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

这个版本的程序依靠 Windows 保存滚动条信息并做边界检查。在 WM_VSCROLL 和 WM_HSCROLL 处理的开始,它获取所有的滚动条信息,根据通知码调整位置,然后调用 SetScrollInfo 设置其位置。程序然后调用 GetScrollInfo。如果该位置超出了 SetScrollInfo 调用的范围,则由 Windows 来纠正该位置,并在 GetScrollInfo 调用中返回正确的值。

SYSMETS3 使用 ScrollWindow 函数在窗口的客户区中滚动信息而不是重画它。虽然该函数很复杂(在新版本的 Windows 中已被更复杂的 ScrollWindowEx 所替代),SYSMETS3 仍以相当简单的方式使用它。函数的第二个参数给出了水平滚动客户区的数值,第三个参数是垂直滚动客户区的数值,单位都是像素。

ScrollWindow 的最后两个参数设置为 NULL,这指出了要滚动整个客户区。Windows 自动把客户区中未被滚动操作覆盖的矩形设为无效。这会产生 WM_PAINT 消息。再也不需要 InvalidateRect 了。注意 ScrollWindow 不是 GDI 函数,因为它不需设备描述表句柄。它是少数几个非 GDI 的 Windows 函数之一,它可以改变窗口的客户区外观。令人感到有些怪但其实很方便的是它随滚动条函数一起记载于文档。

WM_HSCROLL 处理捕获 SB_THUMBPOSITION 通知码并忽略 SB_THUMBTRACK。因而,如果用户在水平滚动条上拖动滚动框,在用户释放鼠标键之前,程序不会水平滚动窗口的内容。

WM_VSCROLL 的方法与之不同:程序捕获 SB_THUMBTRACK 消息,并忽略 SB_THUMBPOSITION。因此,程序随用户在垂直滚动条上拖动滚动框而垂直地滚动内容。这种想法相当完美,但应注意:一旦用户发现程序会立即响应拖动的滚动框,他们就会不断地来回拖动滚动框。幸运的是,现在的快速 PC 可以胜任这种严酷的测试,而在较慢的机器上,则可以考虑对 GetSystemMetrics 使用 SB_SLOWMACHINE 参数作为替换的处理过程。

加快 WM_PAINT 处理的一个方法由 SYSMETS3 演示:WM_PAINT 代码确定无效区域中的行并仅仅重画这些行。当然,代码复杂一些但速度很快。

4.4.4 不用鼠标怎么办

在 Windows 的早期,有大量的用户不喜欢使用鼠标,而且 Windows 自身也不要求必须有鼠标。虽然,没有鼠标的 PC 现在走上了单色显示器和点阵打印机的没落之路,但我仍然建议您编写可以使用键盘来模拟鼠标操作的程序,尤其对于像滚动条这样的基本操作对象更是如此。因为我们的键盘有一组光标移动键,所以应该实现同样的操作。

在下一章,您将学习使用键盘和在 SYSMET3 中增加键盘界面的方法。您可能会注意到,SYSMET3 似乎在通知码等于 SB_TOP 和 SB_BOTTOM 时处理了 WM_VSCROLL 消息。前面已经提到过,窗口过程不从滚动条接收这些消息,所以,目前这是多余的代码。当我们在下一章再次回到这个程序时,您将会明白这样做的原因。

第五章

图形基础

图形设备接口(GDI:Graphics Device Interface)是 Windows 的子系统,它负责在视频显示器和打印机上显示图形。正如您所认为的那样,GDI 是 Windows 非常重要的部分。不但您为 Windows 编写的应用系统在显示视觉信息时只使用 GDI,就连 Windows 本身也只使用 GDI 来显示用户界面对象,诸如菜单、滚动条、图标和鼠标光标。

不幸的是,如果要对 GDI 进行全面的讲述,将需要一整本书——当然不是这本书。在本章中,我只是想向您提供画线和填充区域的基本知识,这对于理解下面几章的 GDI 已经足够了。在后面几章中会讲述 GDI 支持的位图、元文件以及格式化文本。

5.1 GDI 的结构

从程序员的观点来看,GDI 由几百个函数调用和一些相关的数据类型、宏和结构组成。但是在开始讲述这些函数的细节之前,让我们先从宏观上了解一下 GDI 的整体结构。

5.1.1 GDI 原理

Windows 98 和 Microsoft Windows NT 中的图形主要由 GDI32.DLL 动态链接库输出的函数来处理。在 Windows 98 中,这个 GDI32.DLL 实际是利用 16 位 GDI.EXE 动态链接库来执行许多函数。在 Windows NT 中,GDI.EXE 只用于 16 位的程序。

这些动态链接库为您安装的视频显示器和任何打印机调用设备驱动程序中的例程。视频驱动程序访问视频显示器的硬件,打印机驱动程序将 GDI 命令转换为各种打印机能够理解的代码或命令。显然,不同的视频显示适配器和打印机要求不同的设备驱动程序。

因为 PC 兼容机上可以连接许多种不同的视频设备,所以,GDI 的主要目的之一是支持与设备无关的图形。Windows 程序应该能够毫无困难地在 Windows 支持的任意一

种图形输出设备上运行,GDI通过将您的程序和不同输出设备的特性隔离开来的方法来达到这一目的。

图形输出设备分为两大类:光栅设备和矢量设备。大多数PC的输出设备是光栅设备,这意味着它们以点模式来表示图像,这类设备包括视频显示适配器、点阵打印机和激光打印机。矢量设备使用线来绘制图像,通常局限于绘图仪。

许多传统的计算机图形编程都是完全基于矢量的,这意味着使用矢量图形系统的程序是在硬件之上一定层次的抽象。输出设备用像素表示图形,但是程序与编程接口之间并不是用像素进行对话的。您当然可以使用Windows GDI作为一个高层次的矢量绘制系统,同时也可以将它用于比较低层次的像素操作。

从这方面来看,Windows GDI和传统的图形接口语言之间的关系,就如同C和其他编程语言之间的关系一样。C以它在不同操作系统和环境之间的高度可移植性而闻名,然而C也以它允许程序员进行低层次系统调用而闻名,这些调用在其他高级语言中通常是不可能的。正如C有时被认为是一种“高级汇编语言”一样,您可以认为GDI是图形设备硬件之间的一种高层接口。

您已经看到,默认时Windows使用基于像素的坐标系。大多数传统的图形语言使用“虚拟”坐标系,其水平和垂直轴的范围从0~32 767。虽然有些图形语言不让您使用像素坐标,但是Windows GDI允许您使用两者之一(甚至基于物理度量的坐标系)。您可以使用虚拟坐标系以便程序远离硬件,或者也可以使用设备坐标系而完全接近硬件。

某些程序员认为一旦开始了基于像素的程序设计,就抛弃了设备无关性。我们在上一章看到,情况未必如此,其中的诀窍是在与设备无关的方式中使用像素。这要求图形接口语言为程序提供一些方法来确定设备的硬件特征,并进行适当的调节。例如,在SYSMETS程序中,我们根据标准系统字体字符的像素大小来确定屏幕上的文本间距,这种方法允许程序针对分辨率、文本大小和方向比例各不相同的显示适配器进行相应的调节。您将在本章看到一些用于确定显示尺寸的其他方法。

早期,许多用户在一种单色显示器上运行Windows。即使近几年,笔记本计算机还只有灰度级别。为此,GDI的设计保证了您可以在编写一个程序时不必过多地担心色彩问题——也就是说,Windows可以将色彩转换为灰度级别。甚至在今天,Windows 98使用的视频显示已经具有了不同的色彩能力(16色、256色、“高彩色”,以及“真彩色”)。虽然,彩色喷墨打印机的成本已经很低了,但是大多数用户仍然坚持使用黑白打印机。不加区别地使用这些设备是可以的,但是您的程序也应该能决定在某种显示设备上有多少色彩可以使用,从而最好地利用硬件。

当然,就如同您编写C程序时,为了使它在其他计算机上运行而遇到一些微妙的移植性问题一样,您也可能不小心让设备依赖性溜进您的Windows程序,这就是不与硬件完全隔离的代价。您还应该知道Windows GDI的局限。虽然可以在显示器上到处移动图形对象,但GDI通常是一个静态的显示系统,只有有限的动画支持。如果需要为游

戏编写复杂的动画，就应该研究一下 Microsoft DirectX，它提供了您需要的支持。

5.1.2 GDI 函数调用

组成 GDI 的几百个函数调用可以分为几大类：

- **获取(或创建)和释放(或清除)设备描述表的函数** 我们在前面的章节中已经看到过，在绘图时需要设备描述表句柄。GetDC 和 RealseDC 函数让您在非 WM_PAINT 的消息期间来做到这一点，而 BeginPaint 和 EndPaint 函数(虽然在技术上它们是 USER 模块而不是 GDI 模块的一部分)在进行绘图的 WM_PAINT 消息期间使用。我们马上还会介绍有关设备描述表的其他一些函数。
- **获取有关设备描述表信息的函数** 再以第四章中 SYSMETS 程序为例，我们使用 GetTextMetrics 函数来获取有关设备描述表中当前所选字体的尺寸信息。在本章后面，我们将看到一个获取非常广泛的设备描述表信息的 DEVCAPS1 程序。
- **绘图函数** 显然，在所有前提条件都得以满足之后，这些函数是真正重要的部分。在上一章中，我们使用 TextOut 函数在窗口的客户区显示一些文本。我们将看到，其他 GDI 函数还可以让您画线、填充区域。在第十四章和第十五章还会看到如何创建位图图像。
- **设置和获取设备描述表参数的函数** 设备描述表的“属性”决定有关绘图函数如何工作的细节。例如，用 SetTextColor 来指定 TextOut(或者其他文本输出函数)所绘制的文本色彩。在第四章的 SYSMETS 程序中，我们使用 SetTextAlign 来告诉 GDI：TextOut 函数中的文本串的开始位置应该在串的右边而不是默认的左边。设备描述表的所有属性都有默认值，当获取设备描述表时设置这些默认值。对于所有的 Set 函数，都有相应的 Get 函数，以允许您获取当前设备描述表属性。
- **使用 GDI 对象的函数** GDI 在这里变得有点混乱。首先举一个例子：默认时使用 GDI 绘制的所有直线都是实线，并具有一个标准的宽度。您可能希望绘制更细的直线，或者是由一系列的点或短划线组成的直线。这种线的宽度和这种线的线型不是设备描述表的属性，而是一个“逻辑画笔”的特征。您可以通过在 CreatePen、CreatePenIndirect 或 ExtCreatePen 函数中指定这些特征来创建一个逻辑画笔，这些函数返回一个逻辑画笔的句柄(虽然这些函数被认为是 GDI 的一部分，但是和大多数 GDI 函数调用不一样，它们不要求设备描述表的句柄)。要使用这个画笔，就要将画笔句柄选进设备描述表。我们认为，设备描述表中当前选中的画笔就是设备描述表的一个属性。这样，您画任何线都使用这个画笔，然后，您可以取消设备描述表中的画笔选择，并清除画笔对象。清除画笔对象是必要的，因为画笔定义占用了分配的内存空间。除了画笔以外，GDI 对象还用于创建填充封闭区域的刷子、字体、位图，以及 GDI 的其他一些方面。

5.1.3 GDI 图元

在屏幕或打印机上显示的图形类型本身可以被分为几类，通常被称为“图元”，它们是：

- **直线和曲线** 线条是所有矢量图形绘制系统的基础。GDI 支持直线、矩形、椭圆（包括椭圆的子集，也就是我们所说的“圆”）、椭圆圆周上的部分曲线即所谓的“弧”，以及贝塞尔曲线，我们将在本章中分别对它们进行介绍。所有更复杂的曲线可由折线（polyline）代替，折线通过一组非常短的直线来定义一条曲线。线条用设备描述表中选中的当前画笔绘制。
- **填充区域** 当一系列直线或者曲线封闭了一个区域时，该区域可以使用当前 GDI 画刷对象进行填充。这个画刷可以是实心色彩、图案（可以是一系列的水平、垂直或者对角标记），也可以是在区域内垂直或者水平重复的位图图像。
- **位图** 位图是位的矩形数组，这些位对应于显示设备上的像素，它们是光栅图形的基础工具。位图通常用于在视频显示器或者打印机上显示复杂（一般都是真实的）图像。位图还可以用于显示必须很快绘制的小图像，诸如图标、鼠标光标，以及在应用工具栏中出现的按钮等。GDI 支持两种类型的位图——老的（虽然还非常有用）“设备有关”位图，是 GDI 对象；新的（如 Windows 3.0 的）“设备无关”位图（或者 DIB），可以存储在磁盘文件中。第十四章和第十五章讨论位图。
- **文本** 文本的数学味道不像计算机图形的其他方面那样浓。文本和几百年的传统印刷术有关，它被许多印刷工人作为一门艺术看待。因此，文本通常不仅是所有的计算机图形系统中最复杂的部分，而且也是最重要的部分（如果识字还是社会基本要求的话）。用于定义 GDI 字体对象和获取字体信息的数据结构是 Windows 中最庞大的部分之一。从 Windows 3.1 开始，GDI 开始支持 TrueType 字体，该字体是在填充轮廓线基础上创建的，这样的填充轮廓线可由其他 GDI 函数处理。基于兼容性和存储大小的考虑，Windows 98 继续支持旧式的基于位图的字体。我会在第十七章讨论字体。

5.1.4 其他方面

GDI 的其他方面无法这么容易地分类，它们是：

- **映射模式和变换** 虽然默认时以像素为单位进行绘图，但是并非局限于此。GDI 映射模式允许您以英寸（或者甚至以几分之一英寸）、mm 或者任何您想使用的单位来绘图（Windows NT 还支持传统的以 3×3 矩阵表示的“坐标变换”，这允许倾斜和旋转图形对象。不幸的是，在 Windows 98 中不支持坐标变换）。
- **元文件** 元文件是以二进制形式存储的 GDI 命令的集合。元文件主要用于通过剪

贴板传输矢量图形表示。第十八章会讨论元文件。

- **区域** 区域是形状任意的复杂区，通常定义为较简单区域的布尔组合。在 GDI 内部，区域除了存储为最初用来定义区域的线条组合以外，还以一系列扫描线的形式存储。您可以将区域用于绘制轮廓、填充和剪裁。
- **路径** 路径是 GDI 内部存储的直线和曲线的集合。路径可以用于绘图、填充和剪裁，还可以转换为区域。
- **剪裁** 绘图可以限制在客户区的某一部分中，这就是所谓的剪裁。剪裁区域可以是矩形或非矩形，剪裁通常是通过区域或者路径来定义的。
- **调色板** 定制调色板通常限于显示 256 色的显示器。Windows 仅保留这些色彩之中的 20 种供系统使用，您可以改变其他 236 种色彩，以准确显示按位图形式存储的真实图像。第十六章会讨论调色板。
- **打印** 虽然本章限于视频显示，但是您在本章中所学到的全部知识都适用于打印。第十三章会讨论打印。

5.2 设备描述表

在开始绘图之前，让我们比第四章更精确地讨论一下设备描述表。

当您想在一个图形输出设备(诸如屏幕或者打印机)上绘图时，您首先必须获得一个设备描述表(或者 DC)的句柄。将句柄返回给程序时，Windows 就给了您使用设备的权限。然后您在 GDI 函数中将这个句柄作为一个参数，向 Windows 标识您想在其上进行绘图的设备。

设备描述表中包含许多确定 GDI 函数如何在设备上工作的当前“属性”，这些属性允许传递给 GDI 函数的参数只包含起始坐标或者尺寸信息，而不必包含 Windows 在设备上显示对象时需要的所有其他信息。例如，调用 TextOut 时，您只需要在函数中给出设备描述表句柄、起始坐标、文本和文本的长度。您不必指定字体、文本色、文本后面的背景色彩，以及字符间距，因为这些属性都是设备描述表的一部分。当您想改变这些属性之一时，您调用一个可以改变设备描述表中属性的函数，以后针对该设备描述表的 TextOut 调用来使用改变后的属性。

5.2.1 获取设备描述表句柄

Windows 提供了几种获取设备描述表句柄的方法。如果在处理一条消息时获取了设备描述表句柄，应该在退出窗口函数之前释放它(或者删除它)。一旦释放了句柄，它就不再有效了。对于打印机设备描述表句柄，规则就没有这么严格。在第十三章会讨论打印。

最常用的获取并释放设备描述表句柄的方法是，在处理 WM_PAINT 消息时，使用 BeginPaint 和 EndPaint 调用：

```
hdc = BeginPaint (hwnd, &ps) ;
[ other program lines]
EndPaint (hwnd, &ps) ;
```

变量 ps 是类型为 PAINTSTRUCT 的结构,该结构的 hdc 字段是 BeginPaint 返回的设备描述表句柄。PAINTSTRUCT 结构又包含一个名为 rcPaint 的 RECT(矩形)结构,rcPaint 定义一个包围窗口客户区无效范围的矩形。使用从 BeginPaint 获得的设备描述表句柄,只能在这个区域内绘图。BeginPaint 调用使该区域有效。

Windows 程序还可以在处理非 WM_PAINT 消息时获取设备描述表句柄:

```
hdc = GetDC (hwnd) ;
[ other program lines]
ReleaseDC (hwnd, hdc) ;
```

这个设备描述表适用于窗口句柄为 hwnd 的客户区。这些调用与 BeginPaint 和 EndPaint 的组合之间的基本区别是,利用从 GetDC 返回的句柄可以在整个客户区上绘图。当然,GetDC 和 ReleaseDC 不使客户区中任何可能的无效区域变成有效。

Windows 程序还可以获取适用于整个窗口(而不仅限于窗口的客户区)的设备描述表句柄:

```
hdc = GetWindowDC (hwnd) ;
[ other program lines]
ReleaseDC (hwnd, hdc) ;
```

这个设备描述表除了客户区之外,还包括窗口的标题栏、菜单、滚动条和框架(frame)。GetWindowDC 函数很少使用,如果想尝试用一用它,则必须捕获 WM_NCPAINT("非客户绘制")消息, Windows 使用该消息在窗口的非客户区上绘图。

BeginPaint、GetDC 和 GetWindowDC 获得的设备描述表都与视频显示器上的某个特定窗口相关。获取设备描述表句柄的另一个更通用的函数是 CreateDC:

```
hdc = CreateDC (pszDriver, pszDevice, pszOutput, pData) ;
[ other program lines]
DeleteDC (hdc) ;
```

例如,您可以通过下面的调用来获取整个屏幕的设备描述表句柄:

```
hdc = CreateDC (TEXT ("DISPLAY"), NULL, NULL, NULL) ;
```

在窗口之外写东西一般是不合适的,但对于一些不同寻常的应用程序来说,这样做很方便(您还可以通过在调用 GetDC 时使用一个 NULL 参数,从而获取整个屏幕的设备描述表句柄,不过这一事实并没有载入文档)。在第十三章中,我们将使用 CreateDC 函数来获取一个打印机的设备描述表句柄。

有时您只是需要获取关于某设备描述表的一些信息而并不进行任何绘画,在这种

情况下，您可以使用 CreateIC 来获取一个“信息描述表”的句柄，其参数与 CreateDC 函数相同，例如：

```
hdc = CreateIC (TEXT ("DISPLAY"), NULL, NULL, NULL) ;
```

您不能用这个信息环境句柄往设备上写东西。

当使用位图时，获取一个“内存设备描述表”有时是有用的：

```
hdcMem = CreateCompatibleDC (hdc) ;
[other program lines]
DeleteDC (hdcMem) ;
```

您可以将位图选进内存设备描述表，然后使用 GDI 函数在位图上绘画。我将在第十四章讨论这些技术。

前面已经提到过，元文件是一些 GDI 调用的集合，以二进制形式编码。您可以通过获取元文件设备描述表来创建元文件：

```
hdcMeta = CreateMetaFile (pszFilename) ;
[other program lines]
hmf = CloseMetaFile (hdcMeta) ;
```

在元文件设备描述表有效期间，任何用 hdcMeta 所做的 GDI 调用都变成元文件的一部分而不会显示。在调用 CloseMetaFile 之后，设备描述表句柄变为无效，函数返回一个指向元文件(hmf)的句柄。我会在第十八章讨论元文件。

5.2.2 获取设备描述表信息

一个设备描述表通常是指一个物理显示设备，如视频显示器和打印机。通常，您需要获取有关该设备的信息，其中包括显示器的显示尺寸(单位为像素或者物理度量)和色彩范围。您可以通过调用 GetDeviceCaps(“获取设备能力”)函数来获取这些信息：

```
iValue = GetDeviceCaps (hdc, iIndex) ;
```

其中，参数 iIndex 取值为 WINGDI.H 头文件中定义的 29 个标识符之一。例如，当 iIndex 的值为 HORZRES 时，将导致 GetDeviceCaps 返回设备的宽度(单位为像素)；当 iIndex 的值为 VERTRES 时，将导致 GetDeviceCaps 返回设备的高度(单位为像素)。如果 hdc 是屏幕设备描述表的句柄，则 GetDeviceCaps 返回打印机显示区域的高度和宽度，它们也是以像素为单位的。

还可以使用 GetDeviceCaps 来确定设备处理各种不同类型的图形的能力，这通常对于处理视频显示并不十分重要，但是对于使用打印设备却是非常重要。例如，大多数笔式绘图仪不能画位图图像，而 GetDeviceCaps 就可以将这一情况告诉您。

5.2.3 DEVCAPS1 程序

程序 5-1 所示的 DEVCAPS1 程序显示了以一个视频显示器的设备描述表为参数时,可以从 GetDeviceCaps 函数中获得的部分信息(该程序的另一个扩充版本 DEVCAPS2 将在第十三章给出,用于获取打印机信息)。

程序 5-1　DEVCAPS1 程序

DEVCAPS1.C

```
/*--------------------------------------------------
     DEVCAPS1.C——Device Capabilities Display Program No. 1
                 (c) Charles Petzold, 1998
  --------------------------------------------------*/

#include <windows.h>

#define NUMLINES ((int) (sizeof devcaps / sizeof devcaps [0]))

struct
{
     int     iIndex ;
     TCHAR * szLabel ;
     TCHAR * szDesc ;
}
devcaps [] =
{
     HORZSIZE,      TEXT ("HORZSIZE"),     TEXT ("Width in millimeters:"),
     VERTSIZE,      TEXT ("VERTSIZE"),     TEXT ("Height in millimeters:"),
     HORZRES,       TEXT ("HORZRES"),      TEXT ("Width in pixels:"),
     VERTRES,       TEXT ("VERTRES"),      TEXT ("Height in raster lines:"),
     BITSPIXEL,     TEXT ("BITSPIXEL"),    TEXT ("Color bits per pixel:"),
     PLANES,        TEXT ("PLANES"),       TEXT ("Number of color planes:"),
     NUMBRUSHES,    TEXT ("NUMBRUSHES"),   TEXT ("Number of device brushes:"),
     NUMPENS,       TEXT ("NUMPENS"),      TEXT ("Number of device pens:"),
     NUMMARKERS,    TEXT ("NUMMARKERS"),   TEXT ("Number of device markers:"),
     NUMFONTS,      TEXT ("NUMFONTS"),     TEXT ("Number of device fonts:"),
     NUMCOLORS,     TEXT ("NUMCOLORS"),    TEXT ("Number of device colors:"),
     PDEVICESIZE,   TEXT ("PDEVICESIZE"),  TEXT ("Size of device structure:"),
     ASPECTX,       TEXT ("ASPECTX"),      TEXT ("Relative width of pixel:"),
     ASPECTY,       TEXT ("ASPECTY"),      TEXT ("Relative height of pixel:"),
     ASPECTXY,      TEXT ("ASPECTXY"),     TEXT ("Relative diagonal of pixel:"),
     LOGPIXELSX,    TEXT ("LOGPIXELSX"),   TEXT ("Horizontal dots per inch:"),
     LOGPIXELSY,    TEXT ("LOGPIXELSY"),   TEXT ("Vertical dots per inch:"),
     SIZEPALETTE,   TEXT ("SIZEPALETTE"),  TEXT ("Number of palette entries:"),
     NUMRESERVED,   TEXT ("NUMRESERVED"),  TEXT ("Reserved palette entries:"),
```

```
     COLORRES,          TEXT ("COLORRES"),          TEXT ("Actual color resolution:")
} ;

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("DevCaps1") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Device Capabilities"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}
```

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static int  cxChar, cxCaps, cyChar ;
     TCHAR       szBuffer[10] ;
     HDC         hdc ;
     int         i ;
     PAINTSTRUCT ps ;
     TEXTMETRIC  tm ;

     switch (message)
     {
     case WM_CREATE:
          hdc = GetDC (hwnd) ;

          GetTextMetrics (hdc, &tm) ;
          cxChar = tm.tmAveCharWidth ;
          cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
          cyChar = tm.tmHeight + tm.tmExternalLeading ;

          ReleaseDC (hwnd, hdc) ;
          return 0 ;

     case WM_PAINT:
          hdc = BeginPaint (hwnd, &ps) ;

          for (i = 0 ; i < NUMLINES ; i++)
          {
               TextOut (hdc, 0, cyChar * i,
                        devcaps[i].szLabel,
                        lstrlen (devcaps[i].szLabel)) ;

               TextOut (hdc, 14 * cxCaps, cyChar * i,
                        devcaps[i].szDesc,
                        lstrlen (devcaps[i].szDesc)) ;

               SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;
               TextOut (hdc, 14 * cxCaps + 35 * cxChar, cyChar * i, szBuffer,
                        wsprintf (szBuffer, TEXT ("%5d"),
                             GetDeviceCaps (hdc, devcaps[i].iIndex))) ;

               SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
          }

          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY:
          PostQuitMessage (0) ;
```

```
        return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

可以看到,这个程序非常类似于第四章所示的 SYSMETS1。为了保持代码的短小,我没有包括滚动条,因为我知道信息可以在一屏上显示出来。在 256 色、640×480 的 VGA 上显示的结果如图 5-1 所示。

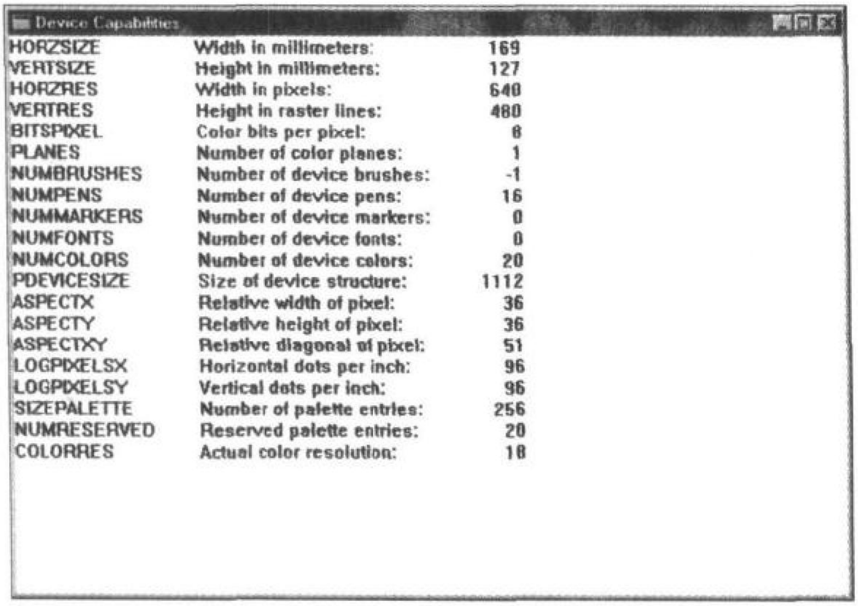


图 5-1 256 色、640×480 VGA 上的 DEVCAPS1 显示

5.2.4 设备的大小

假定要绘制边长为 1 英寸的正方形。您(程序员)或 Windows(操作系统)需要知道视频显示上 1 英寸对应多少像素。使用 GetDeviceCaps 函数能获取有关输出设备物理大小的信息,即视频显示器和打印机。

视频显示器和打印机是两种不同的设备。但也许最不明显的区别是词"分辨率"与设备联系起来的方式。对于打印机,我们经常用"每英寸的点数(dpi)"表示分辨率。例如,大多数激光打印机有 300 或 600 dpi 的分辨率。然而,视频显示器的分辨率是以水平和垂直的总的像素数来表示的,例如,1024×768。大多数人不会告诉您他的打印机在一张纸上水平和垂直打印多少像素或他们的视频显示器上每英寸有多少像素。

在本书中,我用"分辨率"来严格定义每度量单位(一般为英寸)内的像素数。我使用"像素大小"或"像素尺寸"表示设备水平或竖直显示的总像素数。"度量大小"或"度量尺寸"是以英寸或 mm 为单位的设备显示区域的大小。(对于打印机页面,它不是整

个页面，只是可打印的区域。）像素大小除以度量大小就得到分辨率。

现在 Windows 使用的大多数视频显示器的屏幕都是宽比高多 33%。这就表示纵横比为 1.33:1 或 4:3（一般写法）。历史上，该比例可追溯到 Thomas Edison 制作电影的年代。它一直作为电影的标准纵横比，直到 1953 年出现各种类型的宽银幕投影仪。电视机屏幕的纵横比也是 4:3。

然而，Windows 应用程序不应假设视频显示器具有 4:3 的纵横比。人们进行文字处理时希望视频显示器与一张纸的长和宽类似。最普通的选择是把 4:3 变为 3:4 显示，把标准显示翻转一下。

如果设备的水平分辨率与垂直分辨率相等，就称设备具有"正方形像素"。现在，Windows 普遍使用的视频显示器都具有正方形像素，但也有例外。（应用程序也不应假设视频显示器总是具有正方形像素。）当第一次引入 Windows 时，标准视频适配器卡是 IBM Color Graphics Adapter(CGA)，它有 640×200 的像素大小；Enhanced Graphics Adapter(EGA)有 640×350 的像素大小；Hercules Graphics Card 有 720×348 的像素大小。所有这些显示卡都使用 4:3 纵横比的显示器，但是水平和垂直像素数的比值都不是 4:3。

运行 Windows 的用户很容易确定视频显示器的像素大小。在"控制面板"中启动"显示"，并选择"设置"选项卡。在标有"屏幕区域"的域中，可以看到这些像素尺寸之一：

- 640×480 像素；
- 800×600 像素；
- 1024×768 像素；
- 1280×1024 像素；
- 1600×1200 像素。

所有这些都是 4:3。（除了 1280×1024 像素大小。这不但有些不好，还有些令人反感。所有这些像素尺寸在 4:3 的显示器上都认为会产生正方形的像素。）

Windows 应用程序可以使用 SM_CXSCREEN 和 SM_CYSCREEN 参数从 GetSystemMetrics 得到像素尺寸。从 DEVCAPS1 程序中您会注意到，程序可以用 HORZRES（水平分辨率）和 VERTRES 参数从 GetDeviceCaps 中得到同样的值。这里"分辨率"指的是像素大小而不是每度量单位的像素数。

这些是设备大小的简单部分，现在开始复杂的部分。

前两个设备能力，HORZSIZE 和 VERTSIZE，文档中称为"以 mm 计的物理屏幕的宽度"及"以 mm 计的物理屏幕的高度"（在/Platform SDK/Graphics 和 Multimedia Services /GDI/Device Contexts/Device Context Reference/Device Context Functions/GetDeviceCaps 中）。这些看起来更像直接的定义。例如，给出视频显示适配器和显示器的接口特性，Windows 如何真正知道显示器的大小呢？如果您有台膝上型计算机（它的视频驱动程序能

知道准确的屏幕大小),并且连接了外部显示器,又是哪种情况呢?如果把视频投影仪连接到计算机上呢?

在 Windows 的 16 位版本中(以及在 Windows NT 中),Windows 为 HORZSIZE 和 VERTSIZE 使用"标准"的显示大小。然而,从 Windows 95 开始,HORZSIZE 和 VERTSIZE 值是从 HORZRES、VERTRES、LOGPIXELSX 和 LOGPIXELSY 值中衍生出来的。这是它的工作方式。

当您在"控制面板"中使用"显示"程序选择显示的像素大小时,也可以选择系统字体的大小。这个选项的原因是用于 640×480 显示的字体在提升到 1024×768 或更大时字太小,而您可能想要更大的系统字体。这些系统字体大小指"显示"程序的"设置"选项卡中的"小字体"和"大字体"。

在传统的排版中,字体的字母大小由"磅"表示。1 磅大约 1/72 英寸,在计算机排版中 1 磅正好为 1/72 英寸。

理论上,字体的磅值是从字体中最高的字符顶部到例如 j、p、q 和 y 等字母下部的字符底部的距离,其中不包括重音符号。例如,在 10 磅的字体中此距离是 10/72 英寸。根据 TEXTMETRIC 结构,字体的磅值等于 tmHeight 字段减去 tmInternalLeading 字段,如图 5-2 所示。(该图与上一章的图 4-3 一样。)

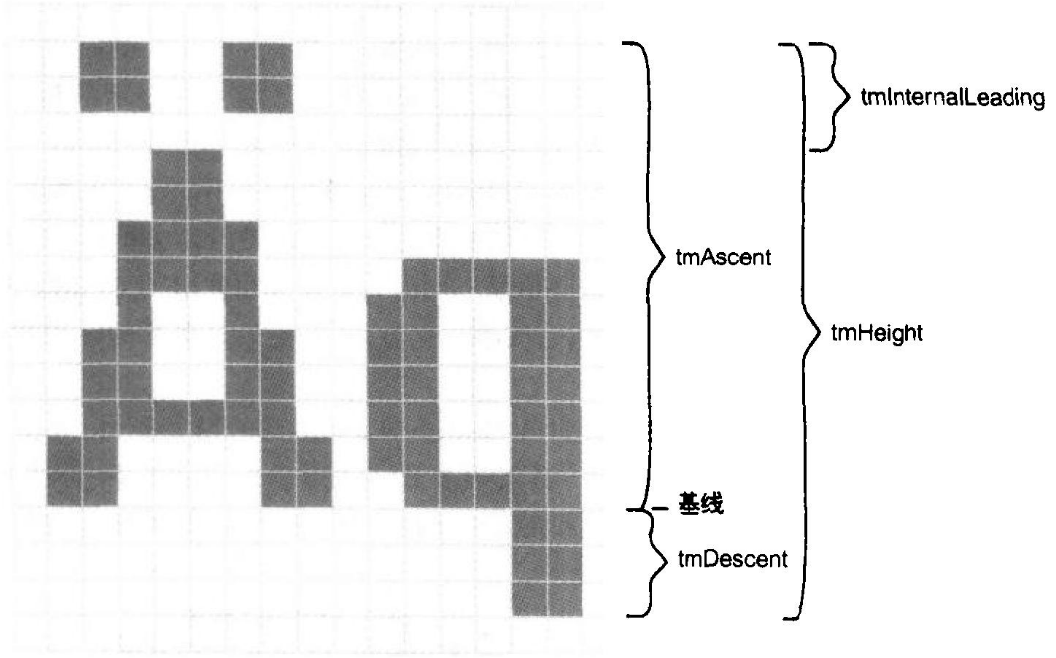


图 5-2 小字体和 TEXTMETRIC 字段

在真正的排版中,字体的磅值与字体字母的实际大小并不正好相等。字体的设计者做出的实际字符比磅值指出的要大一些或小一些。毕竟,字体设计是一种艺术而不

是科学。

TEXTMETRIC 结构的 tmHeight 字段指出文本的连续行在屏幕或打印机上间隔的方式。这也可以用磅来度量。例如,12 磅的行距指出文本连续行的基线应该间隔 12/72(或 1/6)英寸。不应该为 10 磅字体使用 10 磅行距,因为文本的连续行会碰到一起。

10 磅字体读起来很舒服。小于 10 磅的字体不益于长时间阅读。

Windows 系统字体——不考虑是大字体还是小字体,并且也不考虑所选择的视频像素大小——假设是 10 磅字体和 12 磅行距。这听起来很奇怪,如果字体都是 10 磅,为什么还把它们称为大字体和小字体呢?

解答是:当您在"控制面板"的"显示"程序上选择小字体或大字体时,实际上是选择了一个假定的视频显示分辨率,单位是每英寸的点数。当选择小字体时,即要 Windows 假定视频显示分辨率为每英寸 96 点。当选择大字体时,即要 Windows 假定视频显示分辨率为每英寸 120 点。

再看看图 5-2。那是小字体,它基于的显示分辨率为每英寸 96 点。我说过它是 10 磅字体。10 磅即是 10/72 英寸,如果乘以 96 点,每英寸大概就为 13 像素。这即是 tmHeight 减去 tmInternalLeading 的值。行距是 12 磅,或 12/72 英寸,它乘以 96 点,每英寸就为 16 像素。这即是 tmHeight 的值。

下页图 5-3 显示大字体。这是基于每英寸 120 点的分辨率。同样,它是 10 磅字体,10/72 乘以 120 点,每英寸等于 16 像素,即是 tmHeight 减 tmInternalLeading 的值。12 磅行距等于 20 像素,即是 tmHeight 的值。(像第四章一样,再次强调所显示的是实际的度量大小,因此您可以理解它工作的方式。不要在您的程序中对此编写代码。)

在 Windows 程序中,您可以使用 GetDeviceCaps 函数获取用户在"控制面板"的"显示"程序中选择的以每英寸的点数为单位的假定分辨率。要得到这些值(如果视频显示器不具有正方形像素,在理论上这些值是不同的),可以使用索引 LOGPIXELSX 和 LOGPIXELSY。LOGPIXELS 指逻辑像素,它的基本意思是"不是以每英寸的像素数为单位的实际分辨率"。

用 HORZSIZE 和 VERTSIZE 索引从 GetDeviceCaps 得到的设备能力,在文档上称为"物理屏幕的宽度,单位 mm"及"物理屏幕的高度,单位 mm"。因为这些值是从 HORZRES、VERTRES、LOGPIXELSX 和 LOGPIXELSY 值中衍生出来的,所以它们应该称为"逻辑宽度"和"逻辑高度"。公式是

$$\text{水平大小(mm)} = 25.4 \times \frac{\text{水平分辨率(像素)}}{\text{逻辑像素 X(每英寸的点数)}}$$

$$\text{垂直大小(mm)} = 25.4 \times \frac{\text{垂直分辨率(像素)}}{\text{逻辑像素 Y(每英寸的点数)}}$$

常数 25.4 用于把英寸转变为 mm。

这看起来是种倒退和不合逻辑。毕竟,视频显示器是可以用尺子衡量的以 mm 为

单位的大小(至少是近似的)。但是 Windows 98 并不关心这个大小。相反,它以用户选择的显示像素大小和系统字体大小为基础计算以 mm 为单位的显示大小。更改显示的像素大小并根据 GetDeviceCaps 更改度量大小。这有什么意义呢?

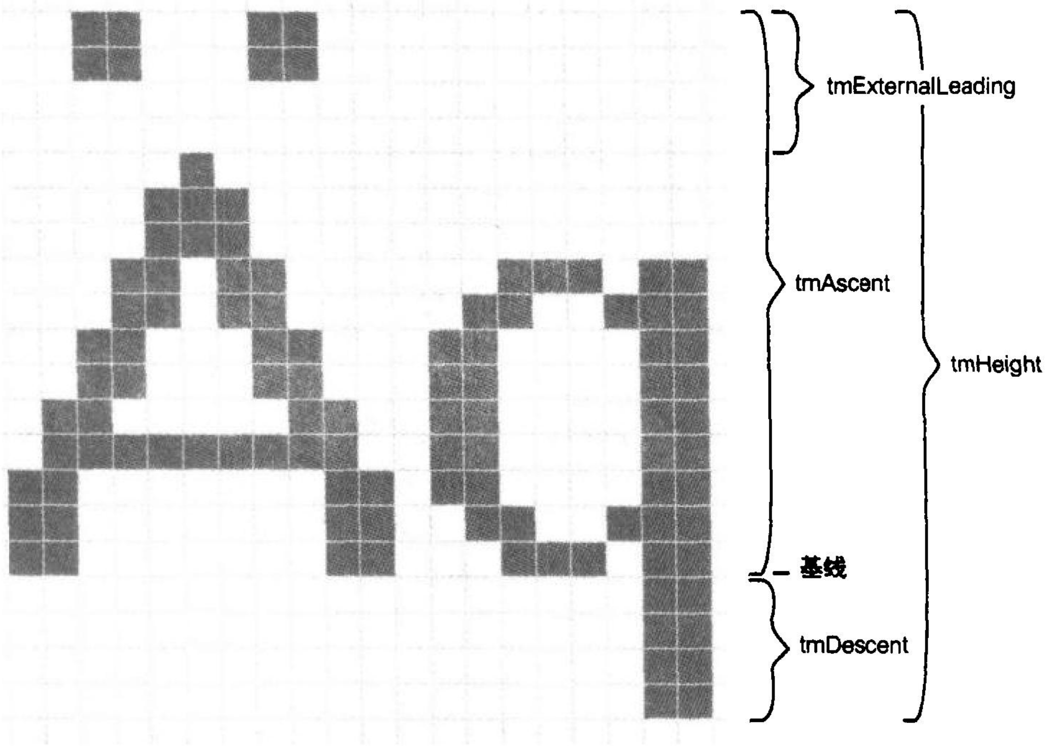


图 5-3 大字体和 FONTMETRIC 字段

这非常有意义。假定有一个 17 英寸的显示器。实际的显示大小大约是 12 英寸乘 9 英寸。假定在最小要求的 640×480 像素大小下运行 Windows。这意味着实际的分辨率是每英寸 53 点。10 磅字体(在纸上便于阅读)在屏幕上从 A 的顶部到 q 的底部只有 7 个像素。这样的字体很难看而且不易读。(可问问那些在老的 Color Graphics Adapter 上运行 Windows 的人们。)

现在,在您的计算机上挂上视频投影仪。投影的视频显示器是 4 英尺宽,3 英尺高。同样的 640×480 像素大小现在是大约每英寸 13 点的分辨率。在这种条件下试图显示 10 磅的字体是很可笑的。

10 磅字体在视频显示器上应是可读的,因为它在打印时是肯定可读的。所以 10 磅字体就成为一个重要的参照。当 Windows 应用程序确保 10 磅屏幕字体为平均大小时,就能够使用 8 磅字体显示较小的文本(仍可读),或用大于 10 磅的字体显示较大的文本。因而,视频分辨率(以每英寸的点数为单位)由 10 磅字体的像素大小来确定是很

有意义的。

然而，在 Windows NT 中，用老的方法定义 HORZSIZE 和 VERTSIZE 值。这种方法与 Windows 的 16 位版本一致。HORZRES 和 VERTRES 值仍然表示水平和垂直像素的数值，LOGPIXELSX 和 LOGPIXELSY 仍然与在"控制面板"的"显示"程序中选择的字体有关。在 Windows 98 中，LOGPIXELSX 和 LOGPIXELSY 的典型值是 96 dpi 和 120 dpi，这取决于您选择的是小字体还是大字体。

在 Windows NT 中的区别是 HORZSIZE 和 VERTSIZE 值固定表示标准显示器大小。对于普通的适配器，获取的 HORZSIZE 和 VERTSIZE 值分别是 320 mm 和 240 mm。这些值是相同的，与选择的像素大小无关。因此，这些值与用 HORZRES、VERTRES、LOGPIXELSX 和 LOGPIXELSY 索引从 GetDeviceCaps 中得到的值不同。然而，可以用前面的公式计算在 Windows 98 下的 HORZSIZE 和 VERTSIZE 值。

如果程序需要实际的视频显示大小该怎么办？也许最好的解决方法是用对话框让用户输入它们。

最后，来自 GetDeviceCaps 的另三个值与视频大小有关。ASPECTX、ASPECTY 和 ASPECTXY 值是每一个像素的相对宽度、高度和对角线大小，四舍五入到整数。对于正方形像素，ASPECTX 和 ASPECTY 值相同。无论如何，ASPECTXY 值应等于 ASPECTX 与 ASPECTY 平方和的平方根，就像直角三角形一样。

5.2.5 关于色彩

如果视频显示适配器仅显示黑色像素和白色像素，则每个像素只需要存储器中的一位。彩色显示器中每个像素需要多个位。位数越多，色彩越多，或者更具体地说，可以同时显示的不同色彩的数目等于 2 的位数次方。

"全色"视频显示器的分辨率是每个像素 24 位——8 位红色、8 位绿色，以及 8 位蓝色。红、绿、蓝即"色光三原色"。混合这三种基本颜色可以生成许多其他的颜色，通过放大镜看显示屏就可以看出来。

"高彩色"显示分辨率是每个像素 16 位——5 位红色、6 位绿色，以及 5 位蓝色。绿色多一位是因为人眼对绿色更敏感一些。

显示 256 种颜色的视频适配器每个像素需要 8 位。然而，这些 8 位的值一般由定义实际颜色的调色板表组织的。我会在第十六章详细地讨论它们。

最后，显示 16 种颜色的显示卡每个像素需要 4 位。这 16 种颜色一般固定分为暗的或浅的红、绿、蓝、青、品红、黄、两种灰色、黑和白。这些 16 位要回溯到老式的 IBM CGA。

仅有在某些古怪的编程任务中才需要知道视频显示卡上的存储器是如何组织的，但是 GetDeviceCaps 使程序员可以确定视频适配器的存储组织，以及它能够表示的色彩

数目,下面的调用返回色彩平面的数目:

```
iPlanes = GetDeviceCaps (hdc, PLANES) ;
```

下面的调用返回每个像素的色彩位数:

```
iBitsPixel = GetDeviceCaps (hdc, BITSPIXEL) ;
```

大多数彩色图形显示设备要么使用多个色彩平面要么每像素有多个色彩位,但是不能同时二者兼用;换句话说,这两个调用必有一个返回 1。视频适配器能够表示的色彩数可以用如下公式来计算:

```
iColors = 1 << (iPlanes * iBitsPixel) ;
```

这个值与用 NUMCOLORS 参数得到的色彩数值可能一样,也可能不一样:

```
iColors = GetDeviceCaps (hdc, NUMCOLORS) ;
```

我提到过,256 色的视频适配器使用色彩调色板。在那种情况下,以 NUMCOLORS 为参数时,GetDeviceCaps 返回由 Windows 保留的色彩数,值为 20,剩余的 236 种颜色可以由 Windows 程序用调色板管理器设置。对于高彩色和全色显示分辨率,带有 NUMCOLORS 参数的 GetDeviceCaps 通常返回 -1,这样就无法得到需要的信息,因此应该使用前面所示的带有 PLANES 和 BITSPIXEL 值的 iColors 公式。

在大多数 GDI 函数调用中,使用 COLORREF 值(只是一个 32 位的无符号长整数)来表示一种色彩。COLORREF 值按照红、绿和蓝色的亮度指定了一种颜色,通常叫做"RGB 色彩"。32 位的 COLORREF 值的设置如图 5-4 所示。

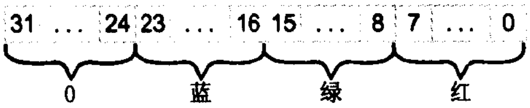


图 5-4　32 位 COLORREF 值

注意最重要的位是标为 0 的 8 位,并且每种基色都指定为一个 8 位的值。理论上,COLORREF 可以指定 2^{24} 或 1600 万种色彩。

这个无符号长整数常常称为一个"RGB 色彩"。Windows 头文件 WINGDI.H 提供了几种使用 RGB 色彩值的宏。RGB 宏要求三个参数分别代表红、绿和蓝值,然后将它们组合为一个无符号长整数:

```
#define RGB(r,g,b) ((COLORREF)(((BYTE)(r) | \
                   ((WORD)((BYTE)(g)) << 8)) | \
                   (((DWORD)(BYTE)(b)) << 16)))
```

注意三个参数的顺序是红、绿和蓝。因此,值

```
RGB (255, 255, 0)
```

是 0x0000FFFF 或黄色(红色和绿色的合成)。当三个参数都设置为 0 时,色彩为黑色;当所有参数都设置为 255 时,色彩为白色。GetRValue、GetGValue 和 GetBValue 宏从 COLORREF 值中抽取出原色值。当您在使用返回 RGB 色彩值的 Windows 函数时,这些宏有时会很方便。

在 16 色或 256 色视频适配器上,Windows 可以使用“抖动”来模拟设备能够显示的颜色之外的色彩。抖动利用了由多种色彩的像素组成的像素图案。可以调用 GetNearestColor 来决定与某一色彩最接近的纯色:

```
crPureColor = GetNearestColor (hdc, crColor) ;
```

5.2.6 设备描述表属性

前面已经提到过,Windows 使用设备描述表来保存控制 GDI 函数在显示器上如何操作的“属性”。例如,在用 TextOut 函数显示文本时,程序员不必指定文本的色彩和字体,Windows 从设备描述表获取这个信息。

程序获取一个设备描述表的句柄时,Windows 用默认值设置所有的属性。(在下一节会看到如何越过这种方式。)表 5-1 列出了 Windows 98 支持的设备描述表属性,程序可以改变或者获取任何一种属性。

表 5-1

设备描述表属性	默认值	修改该值的函数	获取该值的函数
Mapping Mode	MM_TEXT	SetMapMode	GetMapMode
Window Origin	(0,0)	SetWindowOrgEx OffsetWindowOrgEx	GetWindowOrgEx
Viewport Origin	(0,0)	SetViewportOrgEx OffsetViewportOrgEx	GetViewportOrgEx
Window Extents	(1,1)	SetWindowExtEx SetMapMode ScaleWindowExtEx	GetWindowExtEx
Viewport Extents	(1, 1)	SetViewportExtEx SetMapMode ScaleViewportExtEx	GetViewportExtEx
Pen	BLACK_PEN	SelectObject	SelectObject
Brush	WHITE_BRUSH	SelectObject	SelectObject
Font	SYSTEM_FONT	SelectObject	SelectObject
Bitmap	None	SelectObject	SelectObject
Current Position	(0, 0)	MoveToEx LineTo PolylineTo PolyBezierTo	GetCurrentPositionEx

续表

设备描述表属性	默认值	修改该值的函数	获取该值的函数
Background Mode	OPAQUE	SetBkMode	GetBkMode
Background Color	White	SetBkColor	GetBkColor
Text Color	Black	SetTextColor	GetTextColor
Drawing Mode	R2_COPYPEN	SetROP2	GetROP2
Stretching Mode	BLACKONWHITE	SetStretchBltMode	GetStretchBltMode
Polygon Fill Mode	ALTERNATE	SetPolyFillMode	GetPolyFillMode
Intercharacter Spacing	0	SetTextCharacterExtra	GetTextCharacterExtra
Brush Origin	(0, 0)	SetBrushOrgEx	GetBrushOrgEx
Clipping Region	None	SelectObject SelectClipRgn IntersectClipRgn OffsetClipRgn ExcludeClipRect SelectClipPath	GetClipBox

5.2.7 保存设备描述表

通常，在您调用 GetDC 或 BeginPaint 时，Windows 用默认值创建一个新的设备描述表，您对属性所做的一切改变在设备描述表用 ReleaseDC 或 EndPaint 调用释放时，都会丢失。如果您的程序需要使用非默认的设备描述表属性，则您必须在每次获取设备描述表句柄时初始化设备描述表：

```
case WM_PAINT:
    hdc = BeginPaint (hwnd, &ps) ;
    [initialize device context attributes]
    [paint client area of window]
    EndPaint (hwnd, &ps) ;
    return 0 ;
```

虽然在通常情况下这种方法已经很令人满意了，但是您还可能想要在释放设备描述表之后，仍然保存程序中对设备描述表属性所做的改变，以便在下一次调用 GetDC 和 BeginPaint 时它们仍然能够起作用。为此，可在登录窗口类时，将 CS_OWNDC 标志包含为窗口类的一部分：

```
wndclass.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC ;
```

现在，基于这个窗口类所创建的每个窗口都将拥有自己的设备描述表，它一直存在，直到窗口被删除。如果使用了 CS_OWNDC 风格，就只需初始化设备描述表一次，可以在

处理 WM_CREATE 消息期间完成这一操作：

```
case WM_CREATE:
    hdc = GetDC (hwnd) ;
    [initialize device context attributes]
    ReleaseDC (hwnd, hdc) ;
```

这些属性在改变之前一直有效。

CS_OWNDC 风格只影响 GetDC 和 BeginPaint 获得的设备描述表，不影响其他函数(如 GetWindowDC)获得的设备描述表。以前不提倡使用 CS_OWNDC 风格，因为它需要内存；现在，在某些增强图形的 Windows NT 应用程序中，它可以提高性能。即使用了 CS_OWNDC，您仍然应该在退出窗口过程之前释放设备描述表。

某些情况下，您可能想改变某些设备描述表属性，用改变后的属性进行绘图，然后恢复原来的设备描述表。要简化这一过程，可以通过如下调用来保存设备描述表的状态：

```
idSaved = SaveDC (hdc) ;
```

现在，可以改变一些属性，在想要回到调用 SaveDC 前存在的设备描述表时，调用：

```
RestoreDC (hdc, idSaved) ;
```

您可以在调用 RestoreDC 之前调用 SaveDC 数次。

大多数程序员以不同的方式使用 SaveDC 和 RestoreDC。然而，更像汇编语言中的 PUSH 和 POP 指令，当您调用 SaveDC 时，不需要保存返回值：

```
SaveDC (hdc) ;
```

然后，您可以更改某些属性并再次调用 SaveDC。要将设备描述表恢复到一个已经保存的状态，调用：

```
RestoreDC (hdc, -1) ;
```

这就将设备描述表恢复到最近由 SaveDC 函数保存的状态中。

5.3 画点和线

在第一章，我们谈论过 Windows 图形设备接口将图形输出设备的设备驱动程序与计算机连在一起的方式。在理论上，只要提供 SetPixel 和 GetPixel 函数，就可以使用图形设备驱动程序绘制一切东西了。其余的一切都可以使用 GDI 模块中实现的更高级的例程来处理。例如画线时，只需 GDI 调用 SetPixel 数次，并适当地调整 x 和 y 坐标。

在实际情况中，也的确可以仅使用 SetPixel 和 GetPixel 函数进行您需要的任何绘制。您也可以在这些函数的基础上设计出简洁和构造良好的图形编程系统。唯一的问题是性能。如果一个函数通过几次调用才能到达 SetPixel 函数，那么它运行起来会非常

慢。如果一个图形系统画线和进行其他复杂的图形操作是在设备驱动程序的层次上，它就会更有效得多，因为设备驱动程序对完成这些操作的代码进行了优化。此外，一些视频适配器卡包含了图形协处理器，它允许视频硬件自己绘制图形。

5.3.1 写像素

即使 Windows GPI 包含了 SetPixel 和 GetPixel 函数，但很少使用它们。在本书中，仅在第七章的 CONNECT 程序中使用了 SetPixel 函数，仅在第八章的 WHATCLR 程序中使用了 GetPixel 函数。尽管如此，由它们开始来研究图形仍是非常方便。

SetPixel 函数在指定的 x 和 y 坐标以特定的颜色设置像素：

```
SetPixel (hdc, x, y, crColor) ;
```

如同在任何绘图函数中一样，第一个参数是设备描述表的句柄。第二个和第三个参数指明了坐标位置。通常要获得窗口客户区的设备描述表，并且 x 和 y 相对于该客户区的左上角。最后一个参数是 COLORREF 类型，用于指定颜色。如果在函数中指定的颜色在视频显示器上不能被支持，则函数将像素设置为最接近的纯色，并从函数返回该值。

GetPixel 函数返回指定坐标处的像素颜色：

```
crColor = GetPixel (hdc, x, y) ;
```

5.3.2 直线

Windows 可以画直线、椭圆线（椭圆圆周上的曲线）和贝塞尔样条。Windows 98 支持的 7 个画线函数是：

- LineTo 画直线；
- Polyline 和 PolylineTo 画一系列相连的直线；
- PolyPolyline 画多组相连的线；
- Arc 画椭圆线；
- PolyBezier 和 PolyBezierTo 画贝塞尔样条。

另外，Windows NT 还支持 3 种画线函数：

- ArcTo 和 AngleArc 画椭圆线；
- PolyDraw 画一系列相连的线以及贝塞尔样条。

这三个函数 Windows 98 不支持。

在本章的后面我将介绍一些既画线也填充所画图形的封闭区域的函数，这些函数是：

- Rectangle 画矩形;
- Ellipse 画椭圆;
- RoundRect 画带圆角的矩形;
- Pie 画椭圆的一部分,使其看起来像一个扇形;
- Chord 画椭圆的一部分,以呈弓形。

设备描述表的5个属性影响着用这些函数所画线的外观:当前画笔的位置(仅用于LineTo、PolylineTo、PolyBezierTo和ArcTo)、画笔、背景方式、背景色和绘图模式。

画一条直线,必须调用两个函数。第一个函数指定了线的开始点,第二个函数指定了线的终点:

```
MoveToEx (hdc, xBeg, yBeg, NULL) ;
LineTo (hdc, xEnd, yEnd) ;
```

MoveToEx实际上不会画线,它只是设置了设备描述表的"当前位置"属性。然后LineTo函数从当前的位置到它所指定的点画一条直线。当前位置只是用于其他几个GDI函数的开始点。在默认的设备描述表中,当前位置最初设置在点(0,0)。如果在调用LineTo之前没有设置当前位置,那么它将从客户区的左上角开始画线。

一点历史:Windows的16位版本中,用来改变当前位置的函数是MoveTo。该函数只调整三个参数——设备描述表句柄、x和y坐标。函数通过两个16位数拼成的32位无符号长整数返回先前的当前位置。然而,在Windows的32位版本中,坐标是32位的数值,而C的32位版本中又没有定义64位的整数数据类型,因此这种改变意味着MoveTo在其返回值中不再指出先前的当前位置。在实际的编程中,由MoveTo返回的值几乎从来不用,因此就需要一个新函数,这就是MoveToEx。

MoveToEx的最后一个参数是指向POINT结构的指针。从该函数返回后,POINT结构的x和y字段指出了先前的当前位置。如果您不需要这种信息(通常如此),可以简单地如上面的例子所示的那样将最后一个参数设置为NULL。

> **警告**:尽管Windows 98中的坐标值看起来是32位的,实际上却只用到了低16位,坐标值实际上被限制在-32 768~32 767之间。在Windows NT中,使用完整的32位值。

如果您需要当前位置,就可以通过以下调用获得:

```
GetCurrentPositionEx (hdc, &pt) ;
```

其中,pt是POINT结构的。

下面的代码从窗口的左上角开始,在客户区中画一个网格,线与线之间相隔100个像素,其中hwnd是窗口句柄,hdc是设备描述表句柄,而x和y是整数:

```
GetClientRect (hwnd, &rect) ;
```

```
for (x = 0 ; x < rect.right ; x+= 100)
{
     MoveToEx (hdc, x, 0, NULL) ;
     LineTo (hdc, x, rect.bottom) ;
}
for (y = 0 ; y < rect.bottom ; y += 100)
{
     MoveToEx (hdc, 0, y, NULL) ;
     LineTo (hdc, rect.right, y) ;
}
```

虽然用两个函数来画一条直线显得有些麻烦，但是在希望画一组相连的直线时，当前画笔位置属性又会变得很有用。例如，您可能想定义一个包含 5 个点(10 个值)的数组来画一个矩形的边界框：

```
POINT apt[5] = { 100, 100, 200, 100, 200, 200, 100, 200, 100, 100 } ;
```

注意，最后一个点与第一个点相同。现在，只需要使用 MoveToEx 移到第一个点，并对后面的点使用 LineTo：

```
MoveToEx (hdc, apt[0].x, apt[0].y, NULL) ;

for (i = 1 ; i < 5 ; i++)
    LineTo (hdc, apt[i].x, apt[i].y) ;
```

由于 LineTo 从当前位置画到(但不包括)LineTo 函数中给出的点，所以这段代码没有在任何坐标处画两次。虽然在显示器上多输出几次不存在问题，但是在绘图仪上或者在其他绘图方式下(下面马上会讲到)，视觉效果就不太好了。

当您要将数组中的点连接成线时，使用 Polyline 函数要简单得多。下面这条语句画出与上面一段代码相同的矩形：

```
Polyline (hdc, apt, 5) ;
```

最后一个参数是点的数目。我们还可以使用(sizeof (apt)/sizeof (POINT))来表示这个值。Polyline 与一个 MoveToEx 函数后面加几个 LineTo 函数的效果相同，但是，Polyline 既不使用也不改变当前位置。PolylineTo 有些不同，这个函数使用当前位置作为开始点，并将当前位置设置为最后一根线的终点。下面的代码画出与上面所示一样的矩形：

```
MoveToEx (hdc, apt[0].x, apt[0].y, NULL) ;
PolylineTo (hdc, apt + 1, 4) ;
```

您可以对几条线使用 Polyline 和 PolylineTo，不过，当绘制复杂曲线时，这些函数最有用。您使用由几百甚至几千条线组成的极短的线，把它们连在一起就像一条曲线一样。例如，画正弦波就是这样的，程序 5-2 所示的 SINEWAVE 程序显示了如何做到这一点。

程序 5-2 SINEWAVE 程序

SINEWAVE.C

```
/*---------------------------------------
   SINEWAVE.C——Sine Wave Using Polyline
               (c) Charles Petzold, 1998
  ---------------------------------------*/

#include <windows.h>
#include <math.h>

#define NUM    1000
#define TWOPI  (2 * 3.14159)

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("SineWave") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Sine Wave Using Polyline"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
```

```
                         NULL, NULL, hInstance, NULL) ;
     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static int    cxClient, cyClient ;
     HDC           hdc ;
     int           i ;
     PAINTSTRUCT   ps ;
     POINT         apt [NUM] ;

     switch (message)
     {
     case WM_SIZE:
          cxClient = LOWORD (lParam) ;
          cyClient = HIWORD (lParam) ;
          return 0 ;

     case WM_PAINT:
          hdc = BeginPaint (hwnd, &ps) ;

          MoveToEx (hdc, 0,        cyClient / 2, NULL) ;
          LineTo   (hdc, cxClient, cyClient / 2) ;

          for (i = 0 ; i < NUM ; i++)
          {
               apt[i].x = i * cxClient / NUM ;
               apt[i].y = (int) (cyClient / 2 * (1 - sin (TWOPI * i / NUM))) ;
          }

          Polyline (hdc, apt, NUM) ;
          return 0 ;

     case WM_DESTROY:
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

这个程序含有一个1000个POINT结构的数组。随着for循环从0增加到999,结构的x成员设置为从0递增到数值cxClient。结构的y成员设置为一个周期的正弦曲线值,并被放大以充满客户区。整个曲线的绘制仅仅使用了一个Polyline调用。因为Polyline函数是在设备驱动器级别上实现的,因此它要比调用1000次LineTo快得多,结果如图5-5所示。

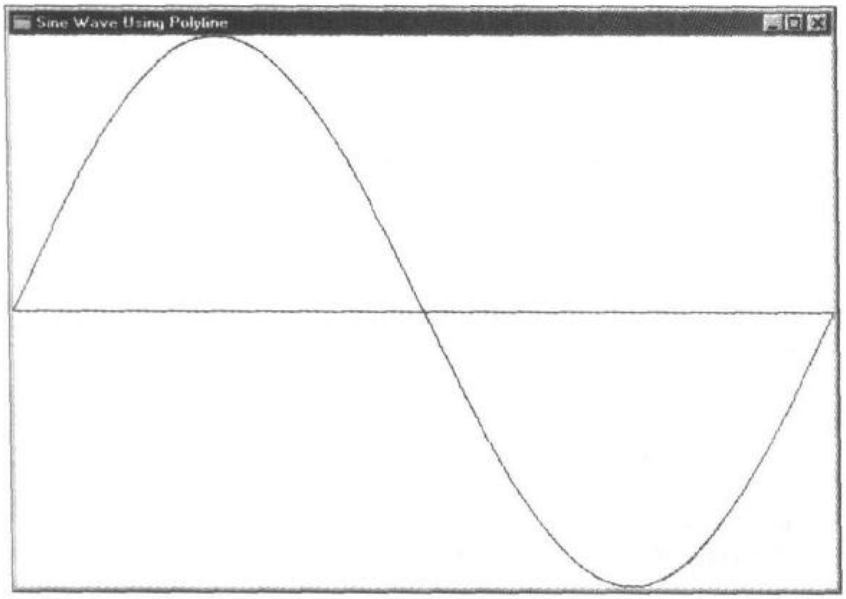


图5-5 SINEWAVE显示

5.3.3 边界框函数

下面我想讨论的是Arc函数,它绘制椭圆曲线。然而,如果不先讨论一下Ellipse函数,那么Arc函数将难以理解;而如果不先讨论Rectangle函数,那么Ellipse函数又将难以理解;而如果讨论Ellipse和Rectangle函数,那么我又会讨论RoundRect、Chord和Pie函数。

问题在于,Rectangle、Ellipse、RoundRect、Chord和Pie函数严格来说不是画线函数。没错,这些函数是在画线,但它们同时又在使用当前的区域填充刷子填充一个封闭区域。这个刷子默认为白色,因此当您第一次使用这些函数时,您可能不会注意到它们不只是画线。严格地说,这些函数属于后面关于"填充区域"的小节,不过,我还是在这里讨论它们。

上面提到的函数有一个共性,即它们都是基于一个矩形边界框的。您定义一个包含该对象的框,即"边界框(bounding box)";Windows就在这个框内画出该对象。

这些函数中最简单的就是画一个矩形:

```
Rectangle (hdc, xLeft, yTop, xRight, yBottom) ;
```

点(xLeft, yTop)是矩形的左上角,点(xRight, yBottom)是矩形的右下角。用函数 Rectangle 画出的图形如图 5-6 所示,矩形的边总是平行于显示器的水平边和垂直边。

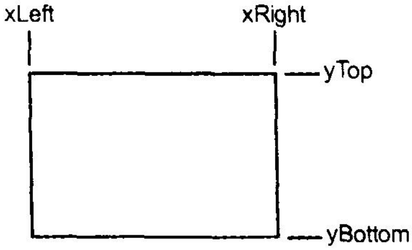


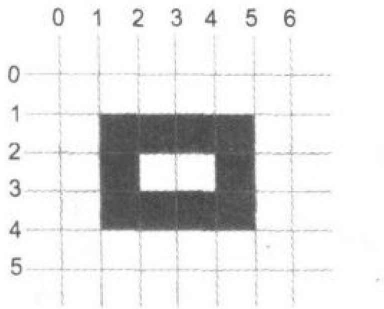
图 5-6 使用 Rectangle 函数画出的图形

以前编过图形程序的程序员熟悉像素偏差的问题。有些图形系统画出的图形包含右坐标和底坐标,而有些则只画到(而不包含)右坐标和底坐标。Windows 采用后一种方法,不过有一种更简单的方法来考虑这个问题。

考虑下面的函数调用:

```
Rectangle (hdc, 1, 1, 5, 4) ;
```

上面我们提到,Windows 在边界框内画图。可以将显示器设想成一个网格,其中,每个像素都在一个网格单元内。边界框画在网格上,然后在边界框内画矩形,下图说明了图形画出来时的样子:



将矩形和客户区左上角分开的区域有 1 个像素宽。

我以前提到过,Rectangle 严格地说不是画线函数,GDI 也填充封闭区域。然而,因为默认用白色填充区域,因此 GDI 填充区域并不明显。

您知道了如何画矩形,也就知道了如何画椭圆,因为它们使用的参数都是相同的:

```
Ellipse (hdc, xLeft, yTop, xRight, yBottom) ;
```

用 Ellipse 函数画出的图形如图 5-7 所示(加上了虚线构成的边界框)。

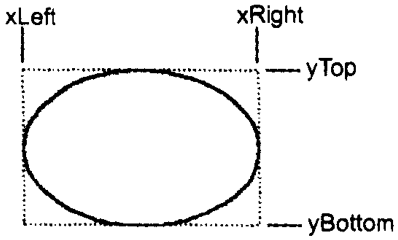


图 5-7 用 Ellipse 函数画出的图形

画圆角矩形的函数使用与函数 Rectangle 及 Ellipse 函数相同的边界框，还包含另外两个参数：

```
RoundRect (hdc, xLeft, yTop, xRight, yBottom,
        xCornerEllipse, yCornerEllipse) ;
```

用这个函数画出的图形如图 5-8 所示。

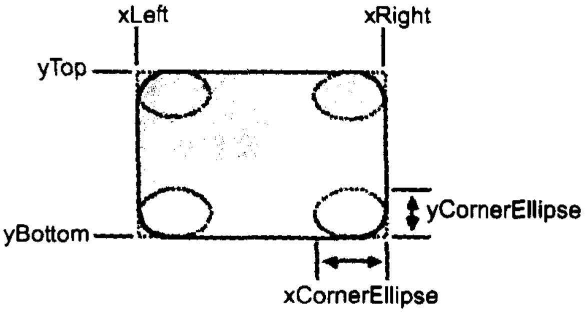


图 5-8 用 RoundRect 函数画出的图形

Windows 使用一个小椭圆来画圆角，这个椭圆的宽为 xCornerEllipse，高为 yCornerEllipse。可以想象这个小椭圆分为了 4 个部分，一个象限一个，每个刚好用在矩形的一个角上。xCornerEllipse 和 yCornerEllipse 的值越大，角就越明显。如果 xCornerEllipse 等于 xLeft 与 xRight 的差，且 yCornerEllipse 等于 yTop 与 yBottom 的差，那么 RoundRect 函数将画出一个椭圆。

在绘制图 5-8 所示的圆角矩形时，用了下面的公式来计算角上椭圆的尺寸。

```
xCornerEllipse = (xRight - xLeft) / 4 ;
yCornerEllipse = (yBottom - yTop) / 4 ;
```

这是一种简单的方法，但是结果看起来有点不对劲儿，因为角的弯曲部分在矩形长的一边要大些。要矫正这一问题，您可以让 xCornerEllipse 与 yCornerEllipse 的值相等。

Arc、Chord 和 Pie 函数都只要相同的参数：

```
Arc   (hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd) ;
Chord (hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd) ;
Pie   (hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd) ;
```

用 Arc 函数画出的线如图 5-9 所示;用 Chord 和 Pie 函数画出的线分别如图 5-10 和图 5-11所示。Windows 用一条假想的线将(xStart, yStart)与椭圆的中心连接,从该线与边界框的交点开始,Windows 按反时针方向,沿着椭圆画一条弧。Windows 还用另一条假想的线将(xEnd,yEnd)与椭圆的中心连接,在该线与边界框的交点处,Windows 停止画弧。

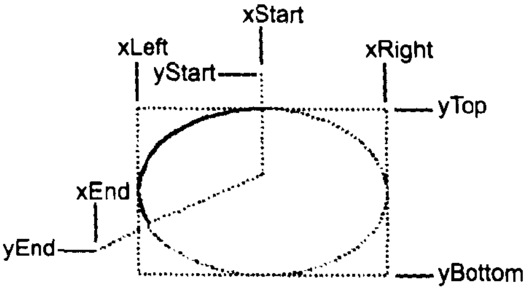


图 5-9 Arc 函数画出的线

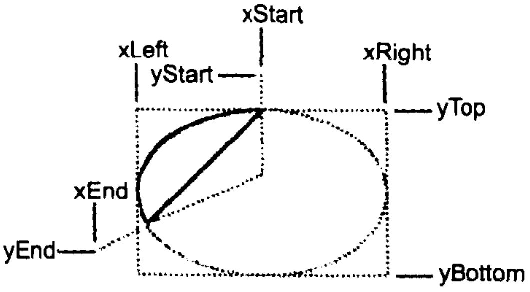


图 5-10 Chord 函数画出的线

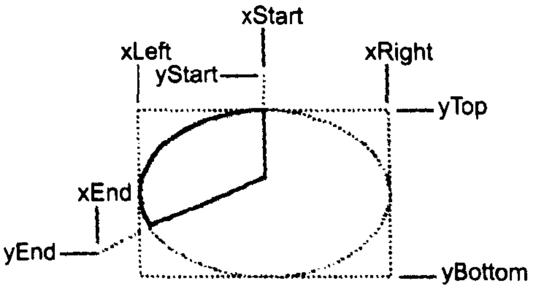


图 5-11 Pie 函数画出的线

对于 Arc 函数,这样就结束了。因为弧只是一条椭圆形的线而已,而不是一个填充

区域。对于 Chord 函数，Windows 连接弧线的端点。而对于 Pie 函数，Windows 将弧的两个端点与椭圆的中心相连接。弦与饼形图的内部以当前刷子填充。

您可能不太明白在 Arc、Chord 和 Pie 函数中开始和结束位置的用法，为什么不简单地在椭圆的周线上指定开始和结束点呢？是的，您可以这么做，但是您将不得不算出这些点。Windows 的方法在不要求这种精确性的条件下完成了相同的工作。

程序 5-3 LINEDEMO 画一个矩形、一个椭圆、一个圆角矩形和两条线段，不过不是按这一顺序。程序表明了定义封闭区域的函数实际上对这些区域进行了填充，因为在椭圆后面的线被遮住了，结果见第 140 页图 5-12 中所示。

程序 5-3　LINEDEMO 程序

LINEDEMO.C

```
/*------------------------------------------------
   LINEDEMO.C -- Line-Drawing Demonstration Program
                 (c) Charles Petzold, 1998
  ------------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("LineDemo") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
```

```
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Line Demonstration"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static int  cxClient, cyClient ;
     HDC         hdc ;
     PAINTSTRUCT ps ;

     switch (message)
     {
     case WM_SIZE:
          cxClient = LOWORD (lParam) ;
          cyClient = HIWORD (lParam) ;
          return 0 ;

     case WM_PAINT:
          hdc = BeginPaint (hwnd, &ps) ;

          Rectangle (hdc,     cxClient / 8,     cyClient / 8,
                          7 * cxClient / 8, 7 * cyClient / 8) ;

          MoveToEx  (hdc,        0,        0, NULL) ;
          LineTo    (hdc, cxClient, cyClient) ;

          MoveToEx  (hdc,        0, cyClient, NULL) ;
          LineTo    (hdc, cxClient,        0) ;

          Ellipse   (hdc,     cxClient / 8,     cyClient / 8,
                          7 * cxClient / 8, 7 * cyClient / 8) ;
```

```
          RoundRect (hdc,    cxClient / 4,      cyClient / 4,
                    3 * cxClient / 4, 3 * cyClient / 4,
                        cxClient / 4,      cyClient / 4) ;

          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY:
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

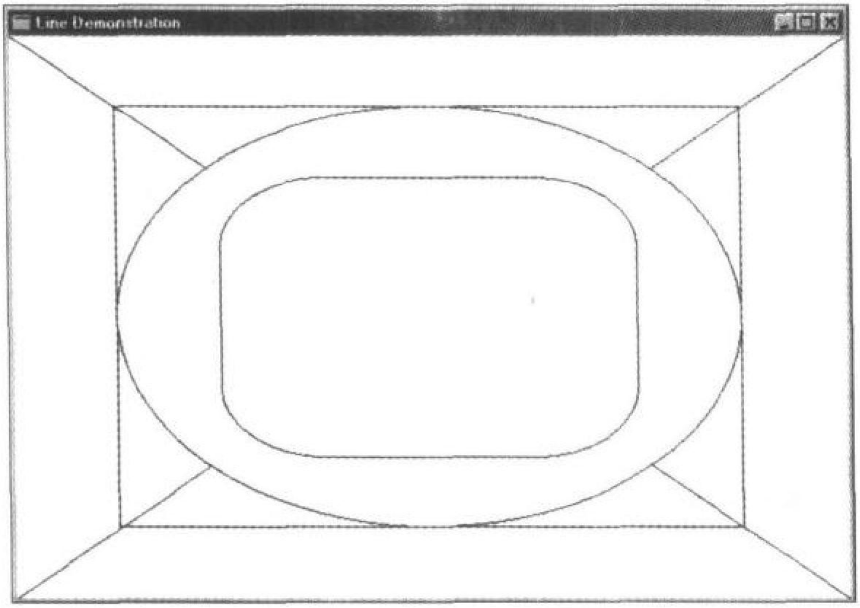


图 5-12 LINEDEMO 显示

5.3.4 贝塞尔样条

"样条"这个词从前指的是一片木头、橡皮或者金属,用来在纸上画曲线。比如说,如果您有一些分开的图上的点,您想要在它们之间画一条曲线(内插或者外插),您首先将这些点描在绘图纸上,然后,将样条定在这些点上,并用铅笔沿着样条绕着这些点弯曲的方向画曲线。

当然,时至今日,样条已经变成了数学公式。有很多种不同的样条,它们各有千秋。贝塞尔样条是计算机程序设计中用得最广的样条之一,它是直到最近才加到操作系统级的图形支持中的。在 60 年代 Renault 汽车公司进行了由手工设计车体(要用到粘土)

到计算机支持的设计的转变。他们需要一些数学工具，而 Pierm Bezier 找到了一套公式，最后表明这套公式对付这样的任务非常有用。

此后，二维的贝塞尔样条成了计算机图形学中最有用的曲线（在直线和椭圆之后）。在 PostScript 中，所有曲线都用贝塞尔曲线表示——椭圆线用贝塞尔曲线来逼近。贝塞尔曲线也用于定义 PostScript 字体的字符轮廓。（TrueType 使用一种更简单更快速的样条。）

一条二维的贝塞尔样条由 4 个点定义——两个端点和两个控制点。曲线的端点在两个端点上，控制点就好像“磁石”一样把曲线从两个端点间的直线处拉走。这可以由一个被称作 BEZIER 的交互式程序做出最好的说明，如程序 5-4 所示。

程序 5-4 BEZIER 程序

BEZIER.C

```
/*--------------------
    BEZIER.C——Bezier Splines Demo
              (c) Charles Petzold, 1998
  --------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
    static TCHAR szAppName[] = TEXT ("Bezier") ;
    HWND        hwnd ;
    MSG         msg ;
    WNDCLASS    wndclass ;

    wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
    wndclass.lpfnWndProc   = WndProc ;
    wndclass.cbClsExtra    = 0 ;
    wndclass.cbWndExtra    = 0 ;
    wndclass.hInstance     = hInstance ;
    wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
    wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
    wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
    wndclass.lpszMenuName  = NULL ;
    wndclass.lpszClassName = szAppName ;

    if (!RegisterClass (&wndclass))
    {
```

```
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Bezier Splines"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

void DrawBezier (HDC hdc, POINT apt[])
{
     PolyBezier (hdc, apt, 4) ;

     MoveToEx (hdc, apt[0].x, apt[0].y, NULL) ;
     LineTo   (hdc, apt[1].x, apt[1].y) ;

     MoveToEx (hdc, apt[2].x, apt[2].y, NULL) ;
     LineTo   (hdc, apt[3].x, apt[3].y) ;
}
LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static POINT  apt[4] ;
     HDC           hdc ;
     int           cxClient, cyClient ;
     PAINTSTRUCT   ps ;

     switch (message)
     {
     case WM_SIZE:
          cxClient = LOWORD (lParam) ;
          cyClient = HIWORD (lParam) ;

          apt[0].x = cxClient / 4 ;
          apt[0].y = cyClient / 2 ;
```

```
          apt[1].x = cxClient / 2 ;
          apt[1].y = cyClient / 4 ;

          apt[2].x =     cxClient / 2 ;
          apt[2].y = 3 * cyClient / 4 ;

          apt[3].x = 3 * cxClient / 4 ;
          apt[3].y =     cyClient / 2 ;

          return 0 ;

     case WM_LBUTTONDOWN:
     case WM_RBUTTONDOWN:
     case WM_MOUSEMOVE:
          if (wParam & MK_LBUTTON || wParam & MK_RBUTTON)
          {
               hdc = GetDC (hwnd) ;

               SelectObject (hdc, GetStockObject (WHITE_PEN)) ;
               DrawBezier (hdc, apt) ;

               if (wParam & MK_LBUTTON)
               {
                    apt[1].x = LOWORD (lParam) ;
                    apt[1].y = HIWORD (lParam) ;
               }

               if (wParam & MK_RBUTTON)
               {
                    apt[2].x = LOWORD (lParam) ;
                    apt[2].y = HIWORD (lParam) ;
               }

               SelectObject (hdc, GetStockObject (BLACK_PEN)) ;
               DrawBezier (hdc, apt) ;
               ReleaseDC (hwnd, hdc) ;
          }
          return 0 ;

     case WM_PAINT:
          InvalidateRect (hwnd, NULL, TRUE) ;

          hdc = BeginPaint (hwnd, &ps) ;

          DrawBezier (hdc, apt) ;

          EndPaint (hwnd, &ps) ;
```

```
        return 0 ;

    case WM_DESTROY:
        PostQuitMessage (0) ;
        return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

由于这个程序要用到一些在第七章才讲的鼠标处理逻辑，所以在这里我不讨论它的内部运作(不过，这也是浅显的)，而是用这个程序来实验性地操纵贝塞尔样条。在这个程序中，两个顶点设置在客户区的上下居中、左右位于 1/4 和 3/4 处的位置；两个控制点可以改变，按住鼠标左键或右键并拖动鼠标可以分别改动两个控制点之一。图 5-13是一个典型的例子。

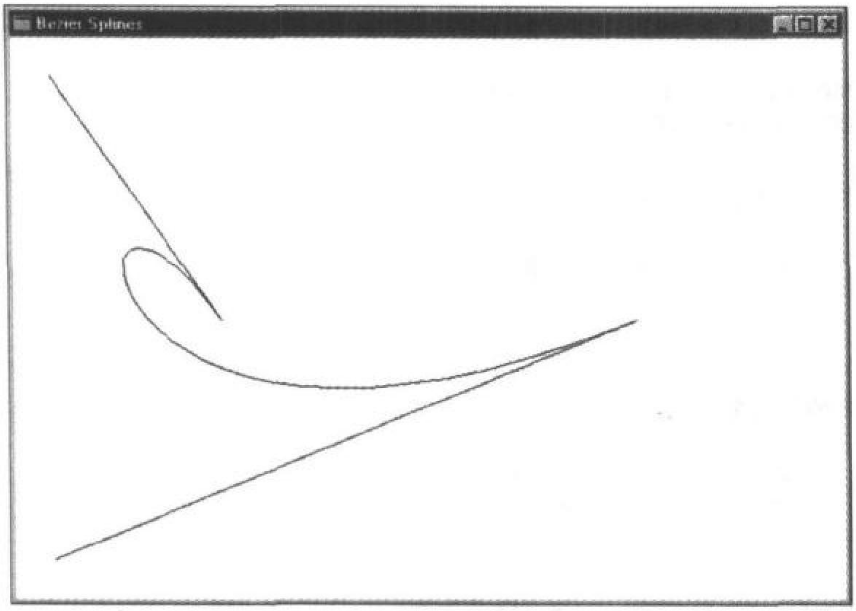


图 5-13 BEZIER 程序的显示

除了贝塞尔样条本身，程序还从第一个控制点向左边的第一个端点(也叫做开始点)画一条直线，并从第二个控制点向右边的端点画一条直线。

由于下面几个特点，贝塞尔样条在计算机辅助设计中非常有用。首先，经过少量练习，就可以把曲线调整到与想要的形状非常接近。

其次，贝塞尔样条非常好控制。对于有的样条种类来说，曲线不经过任何一个定义该曲线的点。贝塞尔样条总是由其两个端点开始和结束的(这是在推导贝塞尔公式时所做的假设之一)。另外，有些形式的样条有奇点，在这些点处曲线趋向无穷远，这在基

于计算机的设计中通常是很不合适的。事实上,贝塞尔曲线总是受限于一个四边形(叫做"凸包"),这个四边形由端点和控制点连接而成。

第三个特点涉及端点和控制点之间的关系。曲线总是与第一个控制点到起点的直线相切,并保持同一方向;同时,也与第二个控制点到终点的直线相切,并保持同一方向。这是用于推导贝塞尔公式时所做的另外两个假设。

第四,贝塞尔样条常常比较具有美感。我知道这是一个主观评价的问题,不过,并非只有我才这样想。

在 32 位的 Windows 版本之前,您必须利用 Polyline 来自己创建贝塞尔样条,并且还需要知道下面的贝塞尔样条的参数方程。起点是(x_0, y_0),终点是(x_3, y_3),两个控制点是(x_1, y_1)和(x_2, y_2),随着 t 的值从 0 到 1 的变化,就可以画出曲线:

$$x(t) = (1-t)^3 x_0 + 3t(1-t)^2 x_1 + 3t^2(1-t)x_2 + t^3 x_3$$

$$y(t) = (1-t)^3 y_0 + 3t(1-t)^2 y_1 + 3t^2(1-t)y_2 + t^3 y_3$$

在 Windows 98 中,您不需要知道这些公式。要画一条或多条连接的贝塞尔样条,只需调用

```
PolyBezier (hdc, apt, iCount) ;
```

或

```
PolyBezierTo (hdc, apt, iCount) ;
```

两种情况下,apt 都是 POINT 结构的数组。对 PolyBezier,前四个点(按照顺序)给出贝塞尔曲线的起点、第一个控制点、第二个控制点和终点。此后的每一条贝塞尔曲线只需给出三个点,因为后一条贝塞尔曲线的起点就是前一条贝塞尔曲线的终点,如此类推。iCount 参数等于 1 加上您所绘制的这些首尾相接曲线条数的三倍。

PolyBezierTo 函数使用当前点作为第一个起点,第一条以及后续的贝塞尔样条都只需要给出三个点。当函数返回时,当前点设置为最后一个终点。

一点提示:在画一系列相连的贝塞尔样条时,只有当第一条贝塞尔曲线的第二个控制点、第一条贝塞尔曲线的终点(也就是第二条曲线的起点)和第二条贝塞尔曲线的第一个控制点线性相关时,也就是说这三个点在同一条直线上时,曲线在连接点处才是光滑的。

5.3.5 使用现有画笔(Stock Pens)

当您调用这一节中讨论的任何画线函数时,Windows 使用设备描述表中当前选中的"画笔"来画线。画笔决定线的色彩、宽度和线型,线型可以是实线、点划线或者虚线,默认设备描述表中画笔为 BLACK_PEN。不管映射方式是什么,这种画笔都画出一个像素宽的黑色实线来。BLACK_PEN 是 Windows 提供的三种现有画笔之一,其他两种是 WHITE

_PEN 和 NULL_PEN,NULL_PEN 什么都不画。您也可以自己定制画笔。

Windows 程序使用句柄来引用画笔。Windows 头文件 WINDEF.H 中包含一个叫做 HPEN 的类型定义,即画笔的句柄,可以定义这个类型的变量(例如 hPen):

```
HPEN hPen ;
```

调用 GetStockObject,可以获得现有画笔的句柄。例如,假设您想使用名为 WHITE_PEN 的现有画笔,可以如下获取画笔的句柄:

```
hPen = GetStockObject (WHITE_PEN) ;
```

现在必须将画笔选进设备描述表:

```
SelectObject (hdc, hPen) ;
```

当前的画笔是白色。在这个调用后,您画的线将使用 WHITE_PEN,直到您将另外一个画笔选进设备描述表或者释放设备描述表句柄为止。

您也可以不显式地定义 hPen 变量,而将 GetStockObject 和 SelectObject 调用合并成一个语句:

```
SelectObject (hdc, GetStockObject (WHITE_PEN)) ;
```

如果想返回到使用 BLACK_PEN 的状态,可以用一条语句获取这种现有画笔的句柄,并将其选进设备描述表:

```
SelectObject (hdc, GetStockObject (BLACK_PEN)) ;
```

SelectObject 的返回值是此调用前设备描述表中的画笔句柄。如果启动一个新的设备描述表并调用

```
hPen = SelectObject (hdc, GetStockobject (WHITE_PEN)) ;
```

则设备描述表中的当前画笔将为 WHITE_PEN,变量 hPen 将会是 BLACK_PEN 的句柄。以后通过调用

```
SelectObject (hdc, hPen) ;
```

能够将 BLACK_PEN 选进设备描述表。

5.3.6 画笔的创建、选择和删除

尽管使用现有画笔非常方便,但却受限于实心的黑画笔、实心的白画笔,或者没有画笔这三种情况。如果想得到更丰富多彩的效果,就必须创建自己的画笔。

这一过程通常是:使用函数 CreatePen 或 CreatePenIndirect 创建一个"逻辑画笔",这

仅仅是对画笔的描述。这些函数返回逻辑画笔的句柄;然后,调用 SelectObject 将画笔选进设备描述表。现在,就可以使用新的画笔来画线了。在任何时候,都只能有一种画笔选进设备描述表。在释放设备描述表(或者在选择了另一种画笔到设备描述表中)之后,就可以调用 DeleteObject 来删除所创建的逻辑画笔了。在删除后,该画笔的句柄就不再有效了。

逻辑画笔是一种"GDI 对象",它是您可以创建的 6 种 GDI 对象之一,其他 5 种是刷子、位图、区域、字体和调色板。除了调色板之外,这些对象都是通过 SelectObject 选进设备描述表的。

在使用画笔等 GDI 对象时,应该遵守以下三条规则:

- ◆ 最后要删除自己创建的所有 GDI 对象;
- ◆ 当 GDI 对象正在一个有效的设备描述表中时,不要删除它;
- ◆ 不要删除现有对象。

这些规则当然是不无道理的,不过有时还挺微妙的。下面我们将举些例子来帮助理解这些规则。

CreatePen 函数的语法形如:

```
hPen = CreatePen (iPenStyle, iWidth, crColor) ;
```

其中,iPenStyle 参数确定画笔是实线、点线还是虚线,该参数可以是 WINGDI.H 头文件中定义的以下标识符,图 5-14 显示了每种线型产生的线类型。

对于 PS_SOLID、PS_NULL 和 PS_INSIDEFRAME 线型,iWidth 参数是画笔的宽度。iWidth 值为 0 则意味着画笔宽度为一个像素。现有画笔是一个像素宽。如果指定的是点划线或者虚线式画笔线型,同时又指定一个大于 1 的物理宽度,那么 Windows 将使用实线画笔来代替。

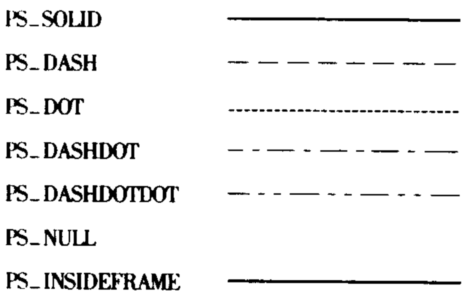


图 5-14 7 种画笔线型

CreatePen 的 crColor 参数是一个 COLORREF 值,它指定画笔的颜色。对于除了 PS_INSIDEFRAME 之外的画笔线型,如果将画笔选入设备描述表中,Windows 会将颜色转换

为设备所能表示的最相近的纯色。PS_INSIDEFRAME 是唯一一种可以使用抖动色的画笔线型，并且只有在宽度大于 1 的情况下才如此。

在与定义一个填充区域的函数一起使用时，PS_INSIDEFRAME 线型还有另外一个奇特之处：对于除了 PS_INSIDEFRAME 以外的所有画笔线型来说，如果用来画边界框的画笔宽度大于 1 个像素，那么画笔将居中对齐在边界框线上，这样边界框线的一部分将位于边界框之外；而对于 PS_INSIDEFRAME 画笔线型来说，整条边界框线都画在边界框之内。

您也可以通过建立一个类型为 LOGPEN（"逻辑画笔"）的结构，并调用 CreatePenIndirect 来创建画笔。如果您的程序使用许多能在源代码中初始化的画笔，那么使用这种方法将有效得多。

要使用 CreatePenIndirect，首先定义一个 LOGPEN 类型的结构：

```
LOGPEN logpen ;
```

此结构有三个成员：lopnStyle（无符号整数或 UINT）是画笔线型，lopnWidth（POINT 结构）是按逻辑单位度量的画笔宽度，lopnColor（COLORREF）是画笔颜色。Windows 只使用 lopnWidth 结构的 x 值作为画笔宽度，而忽略 y 值。

将结构的地址传递给 CreatePenIndirect 结构就可以创建画笔了：

```
hPen = CreatePenIndirect (&logpen) ;
```

注意，CreatePen 和 CreatePenIndirect 函数不需要设备描述表句柄作为参数。这些函数创建与设备描述表没有联系的逻辑画笔。直到调用 SelectObject 之后，画笔才与设备描述表发生联系。因此，可以对不同的设备（如屏幕和打印机）使用相同的逻辑画笔。

下面是创建、选择和删除画笔的一种方法。假设您的程序使用三种画笔——一种宽度为 1 的黑画笔、一种宽度为 3 的红画笔和一种黑色点式画笔，您可以先定义三个变量来存放这些画笔的句柄：

```
static HPEN hPen1, hPen2, hPen3 ;
```

在处理 WM_CREATE 期间，您可以创建这三种画笔：

```
hPen1 = CreatePen (PS_SOLID, 1, 0) ;
hPen2 = CreatePen (PS_SOLID, 3, RGB (255, 0, 0)) ;
hPen3 = CreatePen (PS_DOT, 0, 0) ;
```

在处理 WM_PAINT 期间，或者是在拥有一个设备描述表有效句柄的任何时间里，您都可以将这三个画笔之一选进设备描述表并用它来画线：

```
SelectObject (hdc, hPen2) ;
[line-drawing functions]
```

```
SelectObject (hdc, hPen1) ;
[ line-drawing functions ]
```

在处理 WM_DESTROY 期间，您可以删除您创建的三种画笔：

```
DeleteObject (hPen1) ;
DeleteObject (hPen2) ;
DeleteObject (hPen3) ;
```

这是创建、选择和删除画笔最直接的方法。但是逻辑画笔需要在整个程序运行期间占用存储，为此，您可能想要在每个 WM_PAINT 消息期间创建画笔，并在调用 EndPaint 之后删除它们（您可以在调用 EndPaint 之前删除它们，但是要小心，不要删除设备描述表中当前选择的画笔）。

您可能还希望随时创建画笔，并将 CreatePen 和 SelectObject 调用组合到同一个语句中：

```
SelectObject (hdc, CreatePen (PS_DASH, 0, RGB (255, 0, 0))) ;
```

现在再开始画线，您将使用一个红色虚线画笔。在画完红色虚线之后，可以删除画笔。坏了！由于没有保存画笔句柄，怎么才能删除这些画笔呢？不要紧，请记住，SelectObject 将返回设备描述表中上一次选择的画笔句柄。所以，您可以通过调用 SelectObject 将 BLACK_PEN 选进设备描述表，并删除从 SelectObject 返回的值：

```
DeleteObject (SelectObject (hdc, GetStockObject (BLACK_PEN))) ;
```

下面是另一种方法，在将新创建的画笔选进设备描述表时，保存 SelectObject 返回的画笔句柄：

```
hPen = SelectObject (hdc, CreatePen (PS_DASH, 0, RGB (255, 0, 0))) ;
```

hPen 是什么呢？如果这是在获取设备描述表之后第一次调用 SelectObject，则 hPen 是 BLACK_PEN 对象的句柄。现在，可以将 hPen 选进设备描述表，并删除所创建的画笔（第二次 SelectObject 调用返回的句柄），只要一条语句即可：

```
DeleteObject (SelectObject (hdc, hPen)) ;
```

如果有一个画笔的句柄，就可以通过调用 GetObject 获取 LOGPEN 结构各个成员的值：

```
GetObject (hPen, sizeof (LOGPEN), (LPVOID) &logpen) ;
```

如果需要当前选进设备描述表的画笔句柄，可以调用：

```
hPen = GetCurrentObject (hdc, OBJ_PEN) ;
```

在第十七章将讨论另一个创建画笔的函数 ExtCreatePen。

5.3.7 填充空隙

使用点式画笔和虚线画笔会产生一个有趣的问题：点和虚线之间的空隙会怎样呢？您所需要的是什么？

空隙的着色取决于设备描述表的两个属性——背景模式和背景颜色。默认背景模式为 OPAQUE，在这种方式下，Windows 使用背景色来填充空隙，默认的背景色为白色。这与许多程序在窗口类中用 WHITE_BRUSH 来擦除窗口背景的做法是一致的。

您可以通过如下调用来改变 Windows 用来填充空隙的背景色：

```
SetBkColor (hdc, crColor) ;
```

与画笔色彩所使用的 crColor 参数一样，Windows 将这里的背景色转换为纯色。可以通过用 GetBkColor 来获取设备描述表中定义的当前背景色。

通过将背景模式转换为 TRANSPARENT，可以阻止 Windows 填充空隙：

```
SetBkMode (hdc, TRANSPARENT) ;
```

此后，Windows 将忽略背景色，并且不填充空隙，可以通过调用 GetBkMode 来获取当前背景模式（TRANSPARENT 或者 OPAQUE）。

5.3.8 绘图方式

设备描述表中定义的绘图方式也影响显示器上所画线的外观。设想画这样一条直线，它的色彩由画笔色彩和画线区域原来的色彩共同决定。设想用同一种画笔在白色表面上画出黑线而在黑色表面上画出白线，而且不用知道表面是什么色彩。这样的功能对您有用吗？通过绘图方式的设置，这些都可以实现。

当 Windows 使用画笔来画线时，它实际上执行画笔像素与目标位置处原来像素之间的某种按位布尔运算。像素间的按位布尔运算叫做“光栅运算”，简记为“ROP”。由于画一条直线只涉及两种像素（画笔和目标），因此这种布尔运算又称为“二元光栅运算”，简记为“ROP2”。Windows 定义了 16 种 ROP2 代码，表示 Windows 组合画笔像素和目标像素的方式。在默认设备描述表中，绘图方式定义为 R2_COPYPEN，这意味着 Windows 只是将画笔像素复制到目标像素，这也是我们通常所熟知的。此外，还有 15 种 ROP2 码。

16 种不同的 ROP2 码是怎样得来的呢？为了示范的需要，我们假设使用单色系统，目标色（窗口客户区的色彩）为黑色（用 0 来表示）或者白色（用 1 来表示），画笔也可以为黑色或者白色。用黑色或者白色画笔在黑色或者白色目标上画图有 4 种组合：白笔与白目标、白笔与黑目标、黑笔与白目标，以及黑笔与黑目标。

画笔在目标上绘制后会得到什么呢？一种可能是不管画笔和目标的色彩，画出的

线总是黑色的，这种绘图方式由 ROP2 代码 R2_BLACK 表示。另一种可能是只有当画笔与目标都为黑色时，画出的结果才是白色，其他情况下画出的都是黑色。尽管这似乎有些奇怪，Windows 还是为这种方式起了一个名字，叫做 R2_NOTMERGEPEN。Windows 执行目标像素与画笔像素的按位“或”运算，然后翻转所得色彩。

表 5-2 显示了所有 16 种 ROP2 绘图方式，表中指示了画笔色彩(P)与目标色彩(D)是如何组合而成结果色彩的。在标有“布尔操作”的那一栏中，用 C 语言的表示法给出了目标像素与画笔像素的组合方式。

表 5-2

画笔(P)：	1	1	0	0		
目标(D)：	1	0	1	0	布尔操作	绘图模式
结果：	0	0	0	0	0	R2_BLACK
	0	0	0	1	~(P|D)	R2_NOTMERGEPEN
	0	0	1	0	~P&D	R2_MASKNOTPEN
	0	0	1	1	~P	R2_NOTCOPYPEN
	0	1	0	0	P&~D	R2_MASKPENNOT
	0	1	0	1	~D	R2_NOT
	0	1	1	0	P∧D	R2_XORPEN
	0	1	1	1	~(P&D)	R2_NOTMASKPEN
	1	0	0	0	P&D	R2_MASKPEN
	1	0	0	1	~(P∧D)	R2_NOTXORPEN
	1	0	1	0	D	R2_NOP
	1	0	1	1	~P|D	R2_MERGENOTPEN
	1	1	0	0	P	R2_COPYPEN(默认)
	1	1	0	1	P|~D	R2_MERGEPENNOT
	1	1	1	0	P|D	R2_MERGEPEN
	1	1	1	1	1	R2_WHITE

可以通过以下调用在设备描述表中设置新的绘图模式：

```
SetROP2(hdc, iDrawMode) ;
```

iDrawMode 参数是表中“绘图模式”一栏中给出的值之一。您可以用函数

```
iDrawMode = GetROP2 (hdc) ;
```

来获取当前绘图方式。设备描述表中的默认设置为 R2_COPYPEN，它用画笔色彩替代目标色彩。在 R2_NOTCOPYPEN 方式下，若画笔为黑色，则画成白色；若画笔为白色，则画成黑色。R2_BLACK 方式下，不管画笔和背景色为何种色彩，总是画成黑色。与此相反，R2_WHITE 方式下总是画成白色。R2_NOP 方式就是“不操作”，让目标保持不变。

现在，我们已经讨论了单色系统。然而，大多数系统是彩色的。在彩色系统中，

Windows 为画笔和目标像素的每个颜色位执行绘图方式的按位操作，并再次使用上表描述的 16 种 ROP2 代码。R2_NOT 绘图方式总是翻转目标色彩来决定线的颜色，而不管画笔的色彩是什么。例如，在青色目标上的线会变成紫色。R2_NOT 方式总是产生可见的画笔，除非画笔在中等灰度的背景上绘图。我将在第七章的 BLOKOUT 程序中演示 R2_NOT 绘图方式的使用。

5.4 绘制填充区域

现在再迈出一步，从画线到画图形。Windows 中 7 个用来画带边缘的填充图形的函数列于表5-3中。

表 5-3

函数	图形
Rectangle	直角矩形
Ellipse	椭圆
RoundRect	圆角矩形
Chord	椭圆周上的弧，两端以弦连接
Pie	椭圆上的饼图
Polygon	多边形
PolyPolygon	多个多边形

Windows 用设备描述表中选择的当前画笔来画图形的边界框，边界框还使用当前背景方式、背景色彩和绘图方式，这跟 Windows 画线时一样。关于直线的一切也适用于这些图形的边界框。

图形以当前设备描述表中选择的刷子来填充。默认情况下，使用现有对象，这意味着图形内部将画为白色。Windows 定义 6 种现有刷子：WHITE_BRUSH、LTGRAY_BRUSH、GRAY_BRUSH、DKGRAY_BRUSH、BLACK_BRUSH 和 NULL_BRUSH（也叫 HOLLOW_BRUSH）。您可以将任何一种现有画刷选入您的设备描述表中，就和您选择一种画笔一样。Windows 将 HBRUSH 定义为刷子的句柄，所以可以先定义一个刷子句柄变量：

```
HBRUSH hBrush ;
```

您可以通过调用 GetStockObject 来获取 GRAY_BRUSH 的句柄：

```
hBrush = GetStockObject (GRAY_BRUSH) ;
```

您可以调用 SelectObject 将它选进设备描述表：

```
SelectObject (hdc, hBrush) ;
```

现在，如果您要画上表中的任一个图形，则其内部将为灰色。

如果您想画一个没有边界框的图形，可以将 NULL_PEN 选进设备描述表：

```
SelectObject (hdc, GetStockObject (NULL_PEN)) ;
```

如果您想画出图形的边界框，但不填充内部，则将 NULL_BRUSH 选进设备描述表：

```
SelectObject (hdc, GetStockobject (NULL_BRUSH) ;
```

您也可以定制画刷，就如同您定制画笔一样。我们将马上谈到这个问题。

5.4.1 Polygon 函数和多边形填充方式

我已经讨论过了前 5 个区域填充函数，Polygon 是第 6 个画带边界框的填充图形的函数，该函数的调用与 Polyline 函数相似：

```
Polygon (hdc, apt, iCount) ;
```

其中，apt 参数是 POINT 结构的一个数组，iCount 是点的数目。如果该数组中的最后一个点与第一个点不同，则 Windows 将会再加一条线，将最后一个点与第一个点连起来（在 Polyline 函数中，Windows 不会这么做）。PolyPolygon 函数如下所示：

```
PolyPolygon (hdc, apt, aiCounts, iPolyCount) ;
```

该函数绘制多个多边形。最后一个参数给出了所画的多边形的个数。对于每个多边形，aiCounts 数组给出了多边形的端点数。apt 数组具有全部多边形的所有点。除返回值以外，PolyPolygon 在功能上与下面的代码相同：

```
for (i = 0, iAccum = 0 ; i < iPolyCount ; i++ )
{
     Polygon (hdc, apt + iAccum, aiCounts[i]) ;
     iAccum += aiCounts[i] ;
}
```

对于 Polygon 和 PolyPolygon 函数，Windows 使用定义在设备描述表中的当前画刷来填充这个带边界的区域。至于填充内部的方式，则取决于多边形填充方式，您可以用 SetPolyFillMode 函数来设置：

```
SetPolyFillMode (hdc, iMode) ;
```

默认情况下，多边形填充方式是 ALTERNATE，但是您可以将它设置为 WINDING。两种方式的区别参见图 5-15 所示。

首先，ALTERNATE 和 WINDING 方式之间的区别很容易察觉。对于 ALTERNATE 方式，您可以设想从一个无穷大的封闭区域内部的点画线，只有假想的线穿过了奇数条边界线时，才填充封闭区域。这就是星的角被填充而中心没被填充的原因。

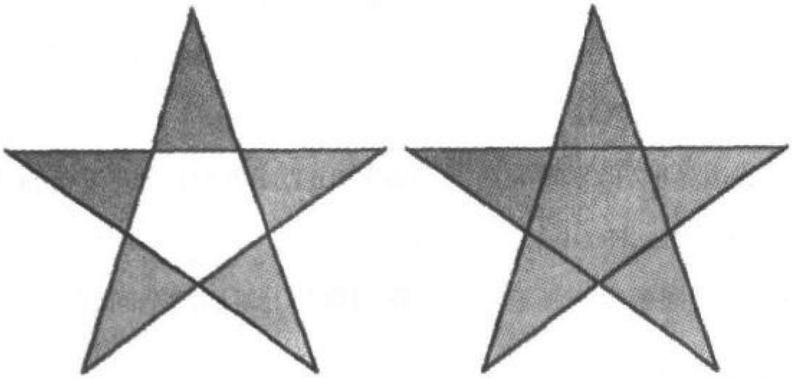


图 5-15　用两种多边形填充方式画出的图：ALTERNATE（左）和 WINDING（右）

五角星的例子使得 WINDING 方式看起来比实际上更简单一些。在绘制单个的多边形时，大多数情况下，WINDING 方式会填充所有封闭的区域。但是也有例外。

在 WINDING 方式下要确定一个封闭区域是否被填充，您仍旧可以设想从那个无穷大的区域画线。如果假想的线穿过了奇数条边界线，区域就被填充，这和 ALTERNATE 方式一样。如果假想的线穿过了偶数条边界线，则区域可能被填充也可能不被填充。如果一个方向（相对于假想线）的边界线数与另一个方向的边界线数不相等，就填充区域。

例如，考虑图 5-16 中的物体。线上的箭头指出了画线的方向。两种方式都会填充三个封闭的 L 形区域，号码为 1～3。号码为 4 和 5 的两个小内部区域，在 ALTERNATE 方式下不会被填充。但是，在 WINDING 方式下，号码为 5 的区域会被填充，因为从区域内必须穿过两条相同方向的线才能到达图形外部。号码为 4 的区域不会被填充，因为必须穿过两条方向相反的线。

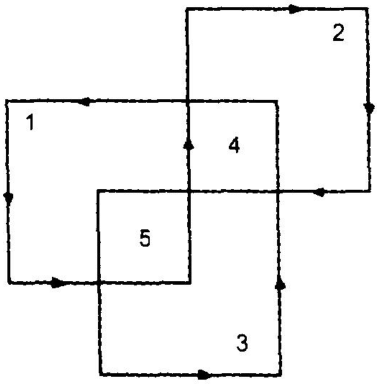


图 5-16　WINDING 方式不能填充所有内部区域的图形

如果您怀疑 Windows 没有这么聪明,那么程序 5-5 ALTWIND 会演示给您看。

程序 5-5 ALTWIND 程序

ALTWIND.C

```
/*----------------------------------------------------
   ALTWIND.C——Alternate and Winding Fill Modes
              (c) Charles Petzold, 1998
   ----------------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("AltWind") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Alternate and Winding Fill Modes"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;
```

```
     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static POINT aptFigure [10] = { 10,70, 50,70, 50,10, 90,10, 90,50,
                                     30,50, 30,90, 70,90, 70,30, 10,30 };
     static int   cxClient, cyClient ;
     HDC          hdc ;
     int          i ;
     PAINTSTRUCT  ps ;
     POINT        apt[10] ;

     switch (message)
     {
     case WM_SIZE:
          cxClient = LOWORD (lParam) ;
          cyClient = HIWORD (lParam) ;
          return 0 ;

     case WM_PAINT:
          hdc = BeginPaint (hwnd, &ps) ;

          SelectObject (hdc, GetStockObject (GRAY_BRUSH)) ;

          for (i = 0 ; i < 10 ; i++)
          {
               apt[i].x = cxClient * aptFigure[i].x / 200 ;
               apt[i].y = cyClient * aptFigure[i].y / 100 ;
          }

          SetPolyFillMode (hdc, ALTERNATE) ;
          Polygon (hdc, apt, 10) ;

          for (i = 0 ; i < 10 ; i++)
          {
               apt[i].x += cxClient / 2 ;
          }

          SetPolyFillMode (hdc, WINDING) ;
```

```
        Polygon (hdc, apt, 10) ;

        EndPaint (hwnd, &ps) ;
        return 0 ;

    case WM_DESTROY:
        PostQuitMessage (0) ;
        return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

图形的坐标(划分为 100×100 个单位)存储在 aptFigure 数组中。这些坐标是以客户区的宽度和高度为基础划分的。程序显示图形两次,一次使用 ALTERNATE 填充方式,另一次使用 WINDING 方式。结果见图 5-17。

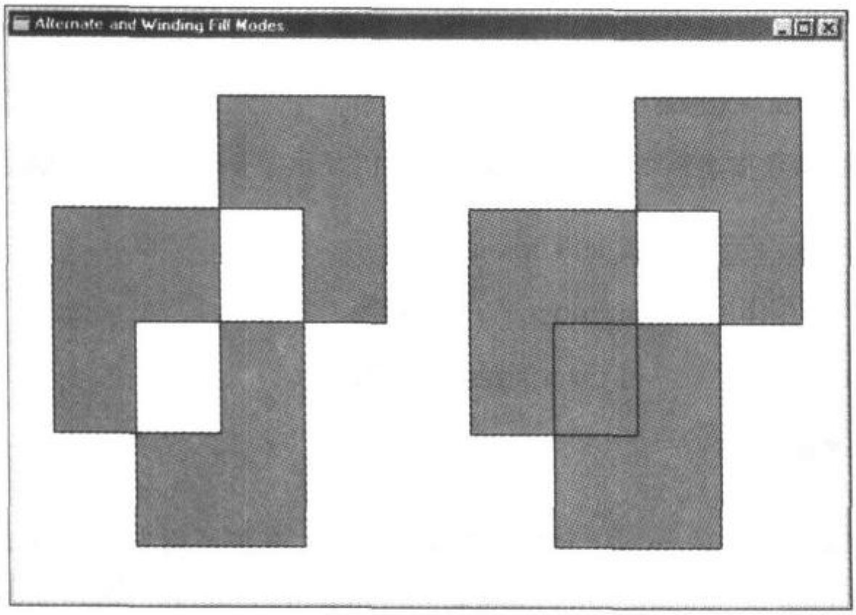


图 5-17 ALTWIND 的显示

5.4.2 用画刷填充内部

Rectangle、RoundRect、Ellipse、Chord、Pie、Polygon 和 PolyPolygon 图形的内部是用选进设备描述表的当前画刷(也称为"模式")来填充的。画刷是一个 8×8 的位图,它水平和垂直地重复使用来填充内部区域。

当 Windows 用抖动的方法来显示多于可从显示器上得到的色彩时,实际上是将画刷用于色彩。在单色系统上,Windows 能够使用黑色和白色像素的抖动创建 64 种不同的灰色,更精确地说,Windows 能够创建 64 种不同的单色画刷。对于纯黑色,8×8 位图

中的所有位均为 0。第一种灰色有一位为 1,第二种灰色有两位为 1,以此类推,直到 8×8 位图中所有位均为 1,这就是白色。在 16 色或 256 色视频系统上,抖动色也是位图,并且可以得到更多的色彩。

Windows 还有 5 个函数,可以让您创建逻辑画刷,然后就可使用 SelectObject 将画刷选进设备描述表。与逻辑画笔一样,逻辑画刷也是 GDI 对象。您创建的所有画刷都必须被删除,但是当它还在设备描述表中时不能将其删除。

下面是创建逻辑画刷的第一个函数:

```
hBrush = CreateSolidBrush (crColor) ;
```

函数中的 Solid 并不是指画刷为纯色。在将画刷选入设备描述表中时,Windows 创建一个抖动色的位图,并为画刷使用该位图。

您还可以使用由水平、垂直或者倾斜的线组成的"影线标记(hatch marks)"来创建画刷,这种风格的画刷对着色条形图的内部和在绘图仪上进行绘图最有用。创建影线画刷的函数为:

```
hBrush = CreateHatchBrush (iHatchStyle, crColor) ;
```

iHatchStyle 参数描述影线标记的外观。图 5-18 显示了 6 种可用的影线标记风格。

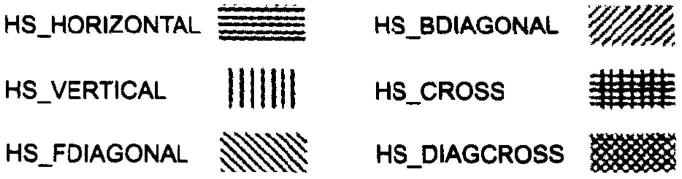


图 5-18 6 种影线画刷风格

CreateHatchBrush 中的 crColor 参数是影线的色彩。在将画刷选进设备描述表时,Windows 将这种色彩转换为与之最相近的纯色。影线之间的区域根据设备描述表中定义的背景方式和背景色来着色。如果背景方式为 OPAQUE,则用背景色(它也被转换为纯色)来填充线之间的空间。在这种情况下,影线和填充色都不能是抖动色。如果背景方式为 TRANSPARENT,则 Windows 只画出影线,不填充它们之间的区域。

您也可以使用 CreatePatternBrush 和 CreateDIBPatternBrushPt 创建自己的基于位图的画刷。

创建逻辑画刷的第 5 个函数包含其他 4 个函数:

```
hBrush = CreateBrushIndirect (&logbrush) ;
```

变量 logbrush 是一个类型为 LOGBRUSH("逻辑画刷")的结构,该结构的三个字段如表 5-4所示,lbStyle 字段的值确定了 Windows 如何解释其他两个字段的值。

表 5-4

lbStyle (UINT)	lbColor (COLORREF)	lbHatch (LONG)
BS_SOLID	画刷的色彩	忽略
BS_HOLLOW	忽略	忽略
BS_HATCHED	影线的色彩	影线画刷风格
BS_PATTERN	忽略	位图的句柄
BS_DIBPATTERNPT	忽略	指向 DIB 的指针

前面我们用 SelectObject 将逻辑画笔选进设备描述表，用 DeleteObject 删除画笔，用 GetObject 来获取逻辑画笔的信息。对于画刷，同样能使用这三个函数。一旦您获取到了画刷句柄，就可以使用 SelectObject 将该画刷选进设备描述表：

```
SelectObject (hdc, hBrush) ;
```

然后，您可以使用 DeleteObject 函数删除所创建的画刷：

```
DeleteObject (hBrush) ;
```

但是，不要删除当前选进设备描述表的画刷。

如果您需要获取画刷的信息，可以调用 GetObject：

```
GetObject (hBrush, sizeof (LOGBRUSH), (LPVOID) &logbrush) ;
```

其中，logbrush 是一个类型为 LOGBRUSH 的结构。

5.5 GDI 映射方式

到目前为止，所有的程序都是相对于客户区的左上角，以像素为单位绘图的。这是默认情况，但不是唯一选择。事实上，“映射方式”是一种几乎影响任何客户区绘图的设备描述表属性。另外有 4 种设备描述表属性——窗口原点、视口原点、窗口范围和视口范围——与映射方式密切相关。

大多数 GDI 绘图函数需要坐标值或大小。例如，下面是 TextOut 函数：

```
TextOut (hdc, x, y, psText, iLength) ;
```

参数 x 和 y 分别表示文本的开始位置。参数 x 是在水平轴上的位置，参数 y 是在垂直轴上的位置，通常用(x,y)来表示这个点。

在 TextOut 中，以及在几乎所有 GDI 函数中，这些坐标值使用的都是一种“逻辑单位”。Windows 必须将逻辑单位转换为“设备单位”，即像素。这种转换是由映射方式、窗口和视口的原点，以及窗口和视口的范围所控制的。映射方式还隐含地给出了 x 轴和 y 轴的指向(orientation)；也就是说，它确定了当您在向显示器的左或者右移动时 x 的

值是增大还是减小,以及在上下移动时 y 的值是增大还是减小。

Windows 定义了 8 种映射方式,它们在 WINGDI.H 中相应的标识符和含义如表 5-5 所示。字 METRIC 和 ENGLISH 指流行的度量系统,磅是印刷的度量单位,约等于 1/72 英寸,但是,在图形编程中我们总假定它正好为 1/72 英寸。“Twip”等于 1/20 磅,也就是 1/1440 英寸。“Isotropic”和“anisotropic”是真正的单词,意思是“各向同性”和“各向异性”。

表 5-5

映射方式	逻辑单位	增加值	
		x 值	y 值
MM_TEXT	像素	右	下
MM_LOMETRIC	0.1 mm	右	上
MM_HIMETRIC	0.01 mm	右	上
MM_LOENGLISH	0.01 in.	右	上
MM_HIENGLISH	0.001 in.	右	上
MM_TWIPS	1/1440 in.	右	上
MM_ISOTROPIC	任意(x = y)	可选	可选
MM_ANISOTROPIC	任意(x!= y)	可选	可选

您可以使用下面的语句来设置映射方式:

```
SetMapMode (hdc, iMapMode) ;
```

其中,iMapMode 是 8 个映射方式标识符之一。您可以通过以下调用获取当前的映射方式:

```
iMapMode = GetMapMode (hdc) ;
```

默认映射方式为 MM_TEXT。在这种映射方式下,逻辑单位与物理单位相同,这样我们可以直接以像素为单位进行操作。在 TextOut 调用中,它看起来像这样:

```
TextOut (hdc, 8, 16, TEXT ("Hello"), 5) ;
```

文本从距离客户区左端 8 像素、上端 16 像素的位置处开始。

如果映射方式设置为 MM_LOENGLISH:

```
SetMapMode (hdc, MM_LOENGLISH) ;
```

则逻辑单位是百分之一。现在,TextOut 调用如下:

```
TextOut (hdc, 50, -100, TEXT ("Hello"), 5) ;
```

文本从距离客户区左端 0.5 英寸、上端 1 英寸的位置处开始。至于 y 坐标前面的

负号,随着我们对映射方式更详细的讨论,也将逐渐清楚。其他映射方式允许程序按照 mm、打印机的点大小,或者任意单位的坐标轴来指定坐标。

如果您认为使用像素进行工作很合适,那么就不要使用默认的 MM_TEXT 方式外的任何映射方式。如果需要以英寸或者 mm 尺寸显示图像,那么可以从 GetDeviceCaps 中获取所需要的信息,自己再进行缩放。其他映射方式都是避免您自己进行缩放的一个方便途径。

虽然您在 GDI 函数中指定的坐标是 32 位的值,但是仅有 Windows NT 能够处理全 32 位。在 Windows 98 中,坐标被限制为 16 位,范围从 - 32 768 到 32 767。一些使用坐标表示矩形的开始点和结束点的 Windows 函数也要求矩形的宽和高小于或者等于 32 767。

5.5.1 设备坐标和逻辑坐标

您也许会问:如果使用 MM_LOENGLISH 映射方式,是不是将会得到以百分之一英寸为单位的 WM_SIZE 消息呢? 绝对不会。Windows 对所有消息(如 WM_MOVE、WM_SIZE 和 WM_MOUSEMOVE),对所有非 GDI 函数,甚至对一些 GDI 函数,永远使用设备坐标。可以这样来考虑:由于映射方式是一种设备描述表属性,所以,只有对需要设备描述表句柄作参数的 GDI 函数,映射方式才会起作用。GetSystemMetrics 不是 GDI 函数,所以它总是以设备单位(即像素)为量度来返回大小的。尽管 GetDeviceCaps 是 GDI 函数,需要一个设备描述表句柄作为参数,但是 Windows 仍然对 HORZRES 和 VERTRES 以设备单位作为返回值,因为该函数的目的之一就是给程序提供以像素为单位的设备大小。

不过,从 GetTextMetrics 调用中返回的 TEXTMETRIC 结构的值是使用逻辑单位的。如果在进行此调用时映射方式为 MM_LOENGLISH,则 GetTextMetrics 将以百分之一英寸为单位提供字符的宽度和高度。在调用 GetTextMetrics 以获取关于字符的宽度和高度信息时,映射方式必须设置成根据这些信息输出文本时所使用的映射方式,这样就可以简化工作。

5.5.2 设备坐标系

Windows 将 GDI 函数中指定的逻辑坐标映射为设备坐标。在讨论以各种不同的映射方式使用逻辑坐标系之前,我们先来看一下 Windows 为视频显示器区域定义的不同的设备坐标系。尽管我们大多数时间在窗口的客户区内工作,但 Windows 在不同的时间使用另外两种设备坐标区域。所有设备坐标系都以像素为单位,水平轴(即 x 轴)上的值从左到右递增,垂直轴(即 y 轴)上的值从上到下递增。

当我们使用整个屏幕时,就根据“屏幕坐标”进行操作。屏幕的左上角为(0,0)点,

屏幕坐标用在 WM_MOVE 消息(对于非子窗口)以及下列 Windows 函数中:CreateWindow 和 MoveWindow(都是对于非子窗口)、GetMessagePos、GetCursorPos、SetCursorPos、GetWindowRect,以及 WindowFromPoint(这不是全部函数的列表)。它们或者是与窗口无关的函数(如两个光标函数),或者是必须相对于某个屏幕点来移动(或者寻找)窗口的函数。如果以 DISPLAY 为参数调用 CreateDC,以获取整个屏幕的设备描述表,则默认情况下 GDI 调用中指定的逻辑坐标将被映射为屏幕坐标。

"整窗口坐标"以程序的整个窗口为基准,如标题栏、菜单、滚动条和窗口框都包括在内。而对于普通窗口,点(0,0)是缩放边框的左上角。整窗口坐标在 Windows 中极少使用,但是如果用 GetWindowDC 获取设备描述表,GDI 函数中的逻辑坐标就会转换为客户区坐标。

第三种坐标系是我们最常使用的"客户区坐标系"。点(0,0)是客户区的左上角。当使用 GetDC 或 BeginPaint 获取设备描述表时,GDI 函数中的逻辑坐标就会默认地转换为客户区坐标。

用函数 ClientToScreen 和 ScreenToClient 可以将客户区坐标转换为屏幕坐标,或者反过来,将屏幕坐标转换为客户区坐标。也可以使用 GetWindowRect 函数获取屏幕坐标下的整个窗口的位置和大小。这三个函数为一种设备坐标转换为另一种提供了足够的信息。

5.5.3 视口和窗口

映射方式定义了 Windows 如何将 GDI 函数中指定的逻辑坐标映射为设备坐标,这里的设备坐标系取决于您用哪个函数来获取设备描述表。要继续讨论映射方式,我们需要一些术语:映射方式用于定义从"窗口"(逻辑坐标)到"视口"(设备坐标)的映射。

"窗口"和"视口"这两个词用得并不恰当。在其他图形界面语言中,视口通常包含有剪裁区域的意思,并且,我们已经用窗口来指程序在屏幕上占据的区域。在这里的讨论中,我们必须把关于这些词的先入之见抛到一边。

"视口"是基于设备坐标(像素)的。通常,视口和客户区相同,但是,如果您已经用 GetWindowDC 或 CreateDC 获取了一个设备描述表,则视口也可以是指整窗口坐标或者屏幕坐标。点(0,0)是客户区(或者整个窗口或屏幕)的左上角,x 的值向右增加,y 的值向下增加。

"窗口"是基于逻辑坐标的,逻辑坐标可以是像素、mm、英寸,或者您想要的任何其他单位。您在 GDI 绘图函数中指定逻辑窗口坐标。

但是在真正的意义上,视口和窗口仅是数学上的概念。对于所有的映射方式,Windows 都用下面两个公式来将窗口(逻辑)坐标转化为视口(设备)坐标:

$$xViewport = (xWindow - xWinOrg) \times \frac{xViewExt}{xWinExt} + xViewOrg$$

$$yViewport = (yWindow - yWinOrg) \times \frac{yViewExt}{yWinExt} + yViewOrg$$

其中,(*xWindow*,*yWindow*)是待转换的逻辑点,(*xViewport*,*yViewport*)是转换后的设备坐标点,更像客户区坐标。

上述两个公式使用了分别指定窗口和视口"原点"的点:(*xWinOrg*,*yWinOrg*)是逻辑坐标的窗口原点;(*xViewOrg*,*yViewOrg*)是设备坐标的视口原点。在默认的设备描述表中,这两个点均被设置为(0,0),但是它们可以改变。此公式意味着,逻辑点(*xWinOrg*,*yWinOrg*)总被映射为设备点(*xViewOrg*,*yViewOrg*)。如果窗口和视口的原点是默认值(0,0),则公式简化为:

$$xViewport = xWindow \times \frac{xViewExt}{xWinExt}$$

$$yViewport = yWindow \times \frac{yViewExt}{yWinExt}$$

这两个公式还使用了两点来指定"范围":(*xWinExt*,*yWinExt*)是逻辑坐标的窗口范围;(*xViewExt*,*yViewExt*)是设备坐标的窗口范围。在多数映射方式中,范围是映射方式所隐含的,不能够改变。每个范围自身没有什么意义,但是视口范围与窗口范围的比是逻辑单位转换为设备单位的换算因子。

例如,当您设置 MM_LOENGLISH 映射方式时,Windows 将 *xViewExt* 设置为某个像素数而将 *xWinExt* 设置为 *xViewExt* 像素占据的一英寸内有几百像素的长度。比值给出了一英寸内有几百个像素的数值。为了提高性能,换算因子表示为整数比而不是浮点数。

范围可以为负,也就是说,逻辑 x 轴上的值不一定非得在向右时增加;逻辑 y 轴上的值不一定非得在向下时增加。

Windows 也能将视口(设备)坐标转换为窗口(逻辑)坐标:

$$xWindow = (xViewport - xViewOrg) \times \frac{xWinExt}{xViewExt} + xWinOrg$$

$$yWindow = (yViewport - yViewOrg) \times \frac{yWinExt}{yViewExt} + yWinOrg$$

Windows 提供了两个函数来让您将设备点转换为逻辑点以及将逻辑点转换为设备点。下面的函数将设备点转换为逻辑点:

```
DPtoLP (hdc, pPoints, iNumber) ;
```

其中,pPoints 是一个指向 POINT 结构数组的指针,而 iNumber 是要转换的点的个数。您会发现这个函数对于将 GetClientRect(它总是使用设备单位)获取的客户区大小转换为逻辑坐标很有用:

```
GetClientRect (hwnd, &rect) ;
```

```
DPtoLP (hdc, (PPOINT) &rect, 2) ;
```

下面的函数将逻辑点转换为设备点：

```
LPtoDP (hdc, pPoints, iNumber) ;
```

5.5.4 处理 MM_TEXT

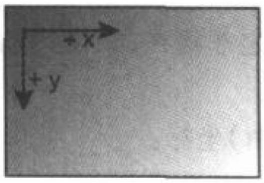
对于 MM_TEXT 映射方式，默认的原点和范围如下所示：

窗口原点：	(0, 0)	可以改变
视口原点：	(0, 0)	可以改变
窗口范围：	(1, 1)	不可改变
视口范围：	(1, 1)	不可改变

视口范围与窗口范围的比例为1，所以不用在逻辑坐标与设备坐标之间进行缩放。上面所给出的公式可以简化为：

$$xViewport = xWindow - xWinOrg + xViewOrg$$
$$yViewport = yWindow - yWinOrg + yViewOrg$$

这种映射方式称为"文本"映射方式，不是因为它对于文本最适合，而是由于轴的方向。我们读文本是从左至右，从上至下，而 MM_TEXT 以同样的方向定义轴上值的增长方向，如下图所示。



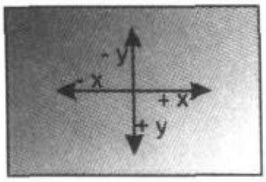
Windows 提供了函数 SetViewportOrgEx 和 SetWindowOrgEx，用来改变视口和窗口的原点，这些函数都具有改变轴的效果，以致(0,0)不再指左上角。一般来说，您会使用 SetViewportOrgEx 或 SetWindowOrgEx 之一，但不会同时使用二者。

我们来看一看这些函数有何效果：如果将视口原点改变为(*xViewOrg*, *yViewOrg*)，则逻辑点(0,0)就会映射为设备点(*xViewOrg*, *yViewOrg*)。如果将窗口原点改变为(*xWinOrg*, *yWinOrg*)，则逻辑点(*xWinOrg*, *yWinOrg*)将会映射为设备点(0,0)，即左上角。不管对窗口和视口原点做什么改变，设备点(0,0)始终是客户区的左上角。

例如，假设客户区为 *cxClient* 个像素宽和 *cyClient* 个像素高。如果想将逻辑点(0,0)定义为客户区的中心，可进行如下调用：

```
SetViewportOrgEx (hdc, cxClient / 2, cyClient / 2, NULL) ;
```

SetViewportOrgEx 的参数总是使用设备单位。现在,逻辑点(0,0)将映射为设备点(*cxClient*/2, *cyClient*/2),而客户区的坐标系变成如下图所示形状。



逻辑 x 轴的范围从 - *cxClient*/2 到 + *cxClient*/2,逻辑 y 轴的范围从 - *cyClient*/2 到 + *cyClient*/2,客户区的右下角为逻辑点(*cxClient*/2, *cyClient*/2)。如果您想从客户区的左上角开始显示文本。则需要使用负坐标:

```
TextOut (hdc, -cxClient / 2, -cyClient / 2, "Hello", 5) ;
```

用下面的 SetWindowOrgEx 语句可以获得与上面使用 SetViewportOrgEx 同样的效果:

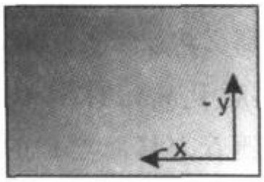
```
SetWindowOrgEx (hdc, -cxClient / 2, -cyClient / 2, NULL) ;
```

SetWindowOrgEx 的参数总是使用逻辑单位。在这个调用之后,逻辑点(- *cxClient*/2, - *cyClient*/2)映射为设备点(0,0),即客户区的左上角。

您不会将这两个函数一起用,除非您知道这么做的结果:

```
SetViewportOrgEx (hdc, cxClient / 2, cyClient / 2, NULL) ;
SetWindowOrgEx (hdc, -cxClient / 2, -cyClient / 2, NULL) ;
```

这意味着逻辑点(- *cxClient*/2, - *cyClient*/2)将映射为设备点(*cxClient*/2, *cyClient*/2),结果是如下图所示的坐标系。



您可以使用下面两个函数获取当前视口和窗口的原点:

```
GetViewportOrgEx (hdc, &pt) ;
GetWindowOrgEx (hdc, &pt) ;
```

其中 pt 是 POINT 结构。由 GetViewportOrgEx 返回的值是设备坐标,而由 GetWindowOrgEx 返回的值是逻辑坐标。

您可能想改变视口或者窗口的原点,以改变窗口客户区内的显示输出——例如,响应用户在滚动条内的输入。但是,改变视口和窗口原点并不能立即改变显示输出,而必须在改变原点之后刷新输出。例如,在第四章的 SYSMETS2 程序中,我们使用了

iVscrollPos 值(垂直滚动条的当前位置)来调整显示输出的 y 坐标：

```
case WM_PAINT:
    hdc = BeginPaint (hwnd, &ps) ;

    for (i = 0 ; i < NUMLINES ; i++)
    {
        y = cyChar * (i - iVscrollPos) ;
        [display text]
    }
    EndPaint (hwnd, &ps) ;
    return 0 ;
```

我们可以使用 SetWindowOrgEx 获得同样的效果。

```
case WM_PAINT:
    hdc = BeginPaint (hwnd, &ps) ;

    SetWindowOrgEx (hdc, 0, cyChar * iVscrollPos) ;

    for (i = 0 ; i < NUMLINES ; i++)
    {
        y = cyChar * i ;
        [display text]
    }
    EndPaint (hwnd, &ps) ;
    return 0 ;
```

现在，TextOut 函数的 y 坐标的计算不需要 iVscrollPos 的值。这意味着您可以将文本输出函数放到一个例程中，不用将 iVscrollPos 值传给该例程，因为我们是通过改变窗口原点来调整文本显示的。

如果您有使用直角坐标系(即笛卡儿坐标系)的经验，那么将逻辑点(0,0)移到客户区的中央(像我们上面所说的那样)的确值得考虑。但是，对于 MM_TEXT 映射方式来说，还存在着一个小小的问题：笛卡儿坐标系中，y 值是随着上移而增加的，而 MM_TEXT 定义为下移时 y 值增加。从这一点来看，MM_TEXT 有点古怪，而下面这 5 种映射方式都使用通常的增值方法。

5.5.5 "度量"映射方式

Windows 包含 5 种以物理尺寸来表示逻辑坐标的映射方式。由于 x 轴和 y 轴的逻辑坐标映射为相同的物理单位，这些映射方式能使您画出不变形的圆和矩形。

这 5 种"度量"映射方式在表 5-6 中列出，按照从低精度到高精度的顺序排列。右边的两列分别给出了以英寸和 mm 为单位时逻辑单位的大小，以便比较。

表 5-6

映射方式	逻辑单位	英寸	mm
MM_LOENGLISH	0.01 英寸	0.01	0.254
MM_LOMETRIC	0.1 mm	0.003 94	0.1
MM_HIENGLISH	0.001 英寸	0.001	0.025 4
MM_TWIPS	1/1400 英寸	0.000 694	0.017 6
MM_HIMETRIC	0.01 mm	0.000 394	0.01

默认窗口及视口的原点和范围如下所示：

窗口原点：	(0, 0)	可以改变
视口原点：	(0, 0)	可以改变
窗口范围：	(?, ?)	不可改变
视口范围：	(?, ?)	不可改变

问号表示窗口和视口的范围依赖于映射方式和设备的分辨率。前面已经提到过，这些范围本身并不重要，但是表示比例时就必须知道。下面是窗口坐标到视口坐标的转换公式：

$$xViewport = (xWindow - xWinOrg) \times \frac{xViewExt}{xWinExt} + xViewOrg$$

$$yViewport = (yWindow - yWinOrg) \times \frac{yViewExt}{yWinExt} + yViewOrg$$

例如，对于 MM_LOENGLISH，Windows 计算的范围如下：

$$\frac{xViewExt}{xWinExt} = 0.01 \text{ 英寸中的水平像素数}$$

$$\frac{-yViewExt}{yWinExt} = 0.01 \text{ 英寸中的垂直像素数}$$

Windows 使用这些来自 GetDeviceCaps 的有用信息设置范围。只是在 Windows 98 和 Windows NT 之间有一点差别。

首先，来看看 Windows 98 是如何做的：假设您使用"控制面板"的"显示"程序选择了 96 dpi 的系统字体。GetDeviceCaps 对于 LOGPIXELSX 和 LOGPIXELSY 索引都将返回值 96。Windows 为视口范围使用这些值并以表 5-7 的方式设置视口和窗口的范围。

这样，对 MM_LOENGLISH 来说，96 除以 100 的比值是 0.01 英寸中的像素数。对 MM_LOMETRIC 来说，96 除以 254 的比值是 0.1 mm 中的像素数。

Windows NT 使用不同的方法设置视口和窗口的范围（与早期 16 位版本的 Windows 一致的方法）。视口范围基于屏幕的像素尺寸。可以使用 HORZRES 和 VERTRES 索引从 GetDeviceCaps 获取这种信息。窗口范围基于假定的显示大小，它是您使用 HORZSIZE

和 VERTSIZE 索引时由 GetDeviceCaps 返回的。我在前面提到过，这些值一般是 320 和 240 mm。如果您将显示器的像素尺寸设置为 1024×768，则表 5-8 就是 Windows NT 报告的视口和窗口范围的值。这些窗口范围表示包含显示器全部宽度和高度的逻辑单位数值。320 mm 宽的屏幕也为 1260 MM_LOENGLISH 单位或 12.6 英寸(320 除以 25.4 mm/英寸)。

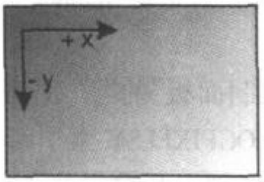
表 5-7

映射方式	视口范围 (x,y)	窗口范围 (x,y)
MM_LOMETRIC	(96, 96)	(254, -254)
MM_HIMETRIC	(96, 96)	(2540, -2540)
MM_LOENGLISH	(96, 96)	(100, -100)
MM_HIENGLISH	(96, 96)	(1000, -1000)
MM_TWIPS	(96, 96)	(1440, -1440)

表 5-8

映射方式	视口范围 (x,y)	窗口范围 (x,y)
MM_LOMETRIC	(1024, -768)	(3 200, 2 400)
MM_HIMETRIC	(1024, -768)	(32 000, 24 000)
MM_LOENGLISH	(1024, -768)	(1 260, 945)
MM_HIENGLISH	(1024, -768)	(12 598, 9 449)
MM_TWIPS	(1024, -768)	(18 142, 13 606)

范围中，y 前面的负号表示改变了轴的方向。对于这 5 种映射方式，y 值随上升而增加，然而注意默认的窗口和视口原点均为(0,0)。这个事实有一个有趣的结果。当一开始改变为 5 种映射方式之一时，坐标系如下图所示：



要想在客户区显示任何东西，必须使用负的 y 值。例如下面的代码：

```
SetMapMode (hdc, MM_LOENGLISH) ;
TextOut (hdc, 100, -100, "Hello", 5) ;
```

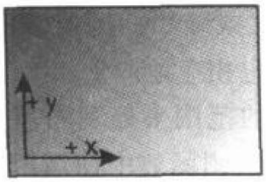
将把文本显示在距离客户区左边和上边各 1 英寸的地方。

为了使自己保持头脑清醒，您可能想避免这样做。一种解决办法是将逻辑的(0,0)

点设为客户区的左下角，您可以通过调用 SetViewportOrgEx 来完成(假设 cyClient 是以像素为单位的客户区的高度)：

```
SetViewportOrgEx (hdc, 0, cyClient, NULL) ;
```

此时的坐标系如下图所示：

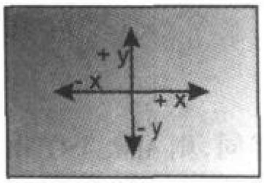


这是直角坐标系的右上象限。

另一种方法是将逻辑(0,0)点设为客户区的中心：

```
SetViewportOrgEx (hdc, cxClient / 2, cyClient / 2, NULL) ;
```

此时的坐标系如下图所示：



现在，我们有了一个真正的 4 象限笛卡儿坐标系，在 x 轴和 y 轴上有相等的按英寸、mm 或 twip 计算的逻辑单位。

您还可以使用 SetWindowOrgEx 函数来改变逻辑(0,0)点，但是这稍微困难一些，因为 SetWindowOrgEx 的参数必须使用逻辑单位，先要将(*cxClient*, *cyClient*)用 DPtoLP 函数转换为逻辑坐标。假设变量 pt 是类型为 POINT 的结构，下面的代码将逻辑(0,0)点改变到客户区的中央：

```
pt.x = cxClient ;
pt.y = cyClient ;
DptoLP (hdc, &pt, 1) ;
SetWindowOrgEx (hdc, -pt.x / 2, -pt.y / 2, NULL) ;
```

5.5.6 “自作主张的”映射方式

剩下的两种映射方式为 MM_ISOTROPIC 和 MM_ANISOTROPIC。只有这两种映射方式运行您改变的视口和窗口范围，也就是说可以改变 Windows 用来转换逻辑和设备坐标的换算因子。词“isotropic”的意思是“各向同性”；“anisotropic”的意思是“各向异性”。与

上面所讨论的度量映射方式相似，MM_ISOTROPIC 使用相同的轴，x 轴上的逻辑单位与 y 轴上的逻辑单位的物理尺寸相等。这对您创建纵横比与显示比无关的图像是有帮助的。

MM_ISOTROPIC 与度量映射方式之间的区别是，使用 MM_ISOTROPIC，您可以控制逻辑单位的物理尺寸。如果愿意，您可以根据客户区的大小来调整逻辑单位的物理尺寸，从而使所画的图像总是包含在客户区内，并相应地放大或缩小。例如，第八章的两个钟程序就是各向同性的例子。在您改变窗口大小时，钟也相应地调整。

Windows 程序完全可以通过调整窗口的大小和视口的范围来处理图像大小的变化。因此，不管窗口尺寸怎样变化，Windows 程序都可以在绘图函数中使用相同的逻辑单位。

有时候 MM_TEXT 和度量映射方式称为"全约束"映射方式，这就是说，您不能改变窗口和视口的范围以及 Windows 将逻辑坐标换算为设备坐标的方法。MM_ISOTROPIC 是一种"半约束"的映射方式，Windows 允许您改变窗口和视口范围，但只是调整它们，以便 x 和 y 逻辑单位代表同样的物理尺寸。MM_ANISOTROPIC 映射方式是"无约束"的，您可以改变窗口和视口范围，但是 Windows 不调整这些值。

MM_ISOTROPIC 映射方式

如果想要在使用任意的轴时都保证两个轴上的逻辑单位相同，则 MM_ISOTROPIC 映射方式就是理想的映射方式。这时，具有相同逻辑宽度和高度的矩形显示为正方形，具有相同逻辑宽度和高度的椭圆显示为圆。

当您刚开始将映射方式设置为 MM_ISOTROPIC 时，Windows 使用与 MM_LOMETRIC 同样的窗口和视口范围（但是，不要对此有所依赖）。区别在于，您现在可以调用 SetWindowExtEx 和 SetViewportExtEx 来根据自己的偏好改变范围了，然后，Windows 将调整范围的值，以便两条轴上的逻辑单位有相同的物理距离。

一般说来，您可以用所期望的逻辑窗口的逻辑尺寸作为 SetWindowExtEx 的参数，用客户区的实际宽和高作为 SetViewportExtEx 的参数。Windows 在调整这些范围时，必须让逻辑窗口适应物理窗口，这就有可能导致客户区的一段落到了逻辑窗口的外面。必须在调用 SetViewportExtEx 之前调用 SetWindowExtEx，以便最有效地使用客户区中的空间。

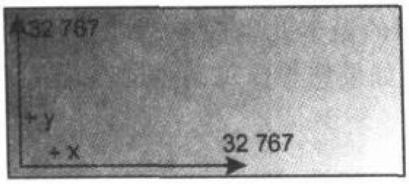
例如，假设您想要一个"传统的"单象限虚拟坐标系，其中(0,0)在客户区的左下角，宽度和高度的范围都是 0～32 767，并且希望 x 和 y 轴的单位具有同样的物理尺寸。以下就是所需的程序：

```
SetMapMode (hdc, MM_ISOTROPIC) ;
SetWindowExtEx (hdc, 32767, 32767, NULL) ;
```

```
SetViewportExtEx (hdc, cxClient, - cyClient, NULL) ;
SetViewportOrgEx (hdc, 0, cyClient, NULL) ;
```

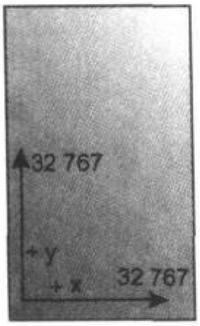
如果其后用 GetWindowExtEx 和 GetViewportExtEx 函数获得了窗口和视口的范围，可以发现，它们并不是先前指定的值。Windows 将根据显示设备的纵横比来调整范围，以便两条轴上的逻辑单位表示相同的物理尺寸。

如果客户区的宽度大于高度(以物理尺寸为准)，Windows 将调整 x 的范围，以便逻辑窗口比客户区视口窄。这样，逻辑窗口将放置在客户区的左边：



Windows 98 不允许在客户区的右边超越 x 轴的范围之外显示任何东西，因为这需要一个大于 16 位能表示的坐标。Windows NT 位坐标使用全 32 位，您可以在超出右边显示一些东西。

如果客户区的高度大于宽度(以物理尺寸为准)，那么 Windows 将调整 y 的范围。这样，逻辑窗口将放置在客户区的下边：

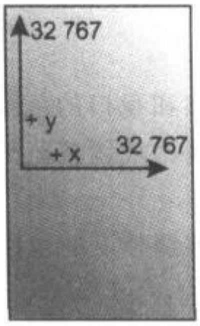


Windows 98 不允许在客户区的顶部显示任何东西。

如果您希望逻辑窗口总是放置在客户区的左上部，那么请将前面给出的代码改为如下所示：

```
SetMapMode (MM_ISOTROPIC) ;
SetWindowExtEx (hdc, 32767, 32767, NULL) ;
SetViewportExtEx (hdc, cxClient, - cyClient, NULL) ;
SetWindowOrgEx (hdc, 0, 32767, NULL) ;
```

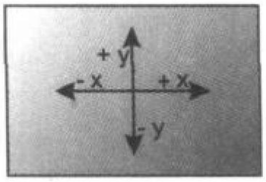
在调用 SetWindowOrgEx 中，我们要求将逻辑点(0,32 767)映射为设备点(0,0)。现在，如果客户区的高大于宽，则坐标系将安排为：



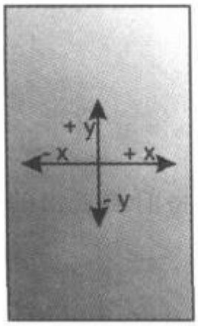
对于时钟程序,您也许想要使用一个4象限的笛卡儿坐标系,4个方向的坐标尺度可以任意指定,(0,0)必须居于客户区的中央。如果您想要每条轴的范围从0到1000,则可以使用以下代码:

```
SetMapMode (hdc, MM_ISOTROPIC) ;
SetWindowExtEx (hdc, 1000, 1000, NULL) ;
SetViewportExtEx (hdc, cxClient / 2, - cyClient / 2, NULL) ;
SetViewportOrgEx (hdc, cxClient / 2, cyClient / 2, NULL) ;
```

如果客户区的宽度大于高度,则逻辑坐标系形如:



如果客户区的高度大于宽度,那么逻辑坐标也会居中:



记住,窗口或者视口范围并不意味着要进行剪裁。在调用GDI函数时,您仍然对以随便地使用小于-1000和大于1000的x和y值。根据客户区的外形,这些点可能看得见,也可能看不见。

在MM_ISOTROPIC映射方式下,可以使逻辑单位大于像素。例如,假设您想要一种映射方式,使点(0,0)显示在屏幕的左上角,y的值向下增长(和MM_TEXT相似),但

是逻辑坐标单位为 1/16 英寸。以下是一种方法：

```
SetMapMode (hdc, MM_ISOTROPIC) ;
SetWindowExtEx (hdc, 16, 16, NULL) ;
SetViewportExtEx (hdc, GetDeviceCaps (hdc, LOGPIXELSX),
                  GetDeviceCaps (hdc, LOGPIXELSY), NULL) ;
```

SetWindowExtEx 函数的参数指出了每一英寸中逻辑单位数。SetViewportExtEx 函数的参数指出了每一英寸中物理单位数(像素)。

然而，这种方法与 Windows NT 中的度量映射方式不一致。这些映射方式使用显示器的像素大小和公制大小。要与度量映射方式保持一致，可以这样做：

```
SetMapMode (hdc, MM_ISOTROPIC) ;
SetWindowExtEx (hdc, 160 * GetDeviceCaps (hdc, HORZSIZE) / 254,
                160 * GetDeviceCaps (hdc, VERTSIZE) / 254, NULL) ;
SetViewportExtEx (hdc, GetDeviceCaps (hdc, HORZRES),
                  GetDeviceCaps (hdc, VERTRES), NULL) ;
```

在这个代码中，视口范围设置为按像素计算的整个屏幕的大小，窗口范围则必须设置为以 1/16 英寸为单位的整个屏幕的大小。GetDeviceCaps 以 HORZRES 和 VERTRES 为参数，返回以毫米为单位的设备尺寸。如果我们使用浮点数，将把毫米数除以 25.4，转换为英寸，然后，再乘以 16 以转换为 1/16 英寸。但是，由于我们使用的是整数，所以先乘以 160，再除以 254。

当然，这种坐标系会使逻辑单位大于物理单位。在设备上输出的所有东西都将映射为按 1/16 英寸增量的坐标值。当然，这样就不能画两条间隔 1/32 英寸的水平直线，因为这样将需要小数逻辑坐标。

MM_ANISOTROPIC：根据需要放缩图像

在 MM_ISOTROPIC 映射方式下设置窗口和视口范围时，Windows 会调整范围，以便两条轴上的逻辑单位具有相同的物理尺度。在 MM_ANISOTROPIC 映射方式下，Windows 不对您所设置的值进行调整，这就是说，MM_ANISOTROPIC 不需要维持正确的纵横比。

使用 MM_ANISOTROPIC 的一种方法是对客户区使用任意坐标，就像我们对 MM_ISOTROPIC 所做的一样。下面的代码将点(0,0)设置为客户区的左下角，x 轴和 y 轴都为 0～32 767：

```
SetMapMode (hdc, MM_ANISOTROPIC) ;
SetWindowExtEx (hdc, 32767, 32767, NULL) ;
SetViewportExtEx (hdc, cxClient, -cyClient, NULL) ;
SetViewportOrgEx (hdc, 0, cyClient, NULL) ;
```

在 MM_ISOTROPIC 方式下，相似的代码导致客户区的一部分在轴的范围之外。但是对

于 MM_ANISOTROPIC，不论其尺度多大，客户区的右上角总是(32 767，32 767)。如果客户区不是正方形的，则逻辑 x 和 y 的单位具有不同的物理尺度。

前一节在 MM_ISOTROPIC 映射方式下，我们讨论了在客户区中画一个类似时钟的图像，x 和 y 轴的范围都是 -1000 ~ +1000。对于 MM_ANISOTROPIC，也可以编写类似的程序：

```
SetMapMode (hdc, MM_ANISOTROPIC) ;
SetWindowExtEx (hdc, 1000, 1000, NULL) ;
SetViewportExtEx (hdc, cxClient / 2, -cyClient / 2, NULL) ;
SetViewportOrgEx (hdc, cxClient / 2, cyClient / 2, NULL) ;
```

与 MM_ANISOTROPIC 方式不同的是，这个钟一般是椭圆形的，而不是圆形的。

另一种使用 MM_ANISOTROPIC 的方法是将 x 和 y 轴的单位固定，但其值不相等。例如，如果有一个只显示文本的程序，您可能想根据单个字符的高度和宽度设置一种粗粒度的坐标：

```
SetMapMode (hdc, MM_ANISOTROPIC) ;
SetWindowExtEx (hdc, 1, 1, NULL) ;
SetViewportExtEx (hdc, cxChar, cyChar, NULL) ;
```

当然，这里假设 cxChar 和 cyChar 分别是那种字体的字符宽度和高度。现在，您可以按字符行和列指定坐标。下面的语句在距离客户区左边三个字符，上边二个字符处显示文本：

```
TextOut (hdc, 3, 2, TEXT ("Hello"), 5) ;
```

如果您使用固定大小的字体时会更加方便，就像下面的 WHATSIZE 程序所示的那样。

当您第一次设置 MM_ANISOTROPIC 映射方式时，它总是继承前面所设置的映射方式的范围，这会很方便。可以认为 MM_ANISOTROPIC 不“锁定”范围；也就是说，它允许您任意改变窗口范围。例如，假设您想用 MM_LOENGLISH 映射方式，因为希望逻辑单位为 0.01 英寸，但您不希望 y 轴的值向上增加，喜欢如 MM_TEXT 那样的方向，即 y 轴的值向下增加，可以使用如下的代码：

```
SIZE size ;
[other program lines]
SetMapMode (hdc, MM_LOENGLISH) ;
SetMapMode (hdc, MM_ANISOTROPIC) ;
GetViewportExtEx (hdc, &size) ;
SetViewportExtEx (hdc, size.cx, -size.cy, NULL) ;
```

我们首先将映射方式设置为 MM_LOENGLISH，然后，通过将映射方式设置为 MM_ANISOTROPIC 让范围可以自由改变。GetViewportExtEx 获取视图并放到一个 SIZE 结构

中，然后，我们使用范围来调用 SetViewportExtEx，只是要将 y 范围取反。

5.5.7 WHATSIZE 程序

Windows 的一点历史：第一篇如何编写 Windows 程序的文章于 1986 年 12 月在《Microsoft Systems Journal》杂志上公开发行。在那篇文章中，示例程序叫做 WSZ（"what size：什么尺寸"），它以像素、英寸和 mm 为单位显示了客户区的大小。那个程序的更简单的版本是 WHATSIZE，如程序 5-6 所示。程序显示了以 5 种度量映射方式显示的窗口客户区的大小。

程序 5-6　WHATSIZE 程序

WHATSIZE.C

```
/*--------------------------------------------
   WHATSIZE.C -- What Size is the Window?
                 (c) Charles Petzold, 1998
  --------------------------------------------*/
#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("WhatSize") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
```

```
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("What Size is the Window?"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}
void Show (HWND hwnd, HDC hdc, int xText, int yText, int iMapMode,
           TCHAR * szMapMode)
{
     TCHAR  szBuffer [60] ;
     RECT   rect ;

     SaveDC  (hdc) ;

     SetMapMode (hdc, iMapMode) ;
     GetClientRect (hwnd, &rect) ;
     DPtoLP (hdc, (PPOINT) &rect, 2) ;

     RestoreDC (hdc, -1) ;

     TextOut (hdc, xText, yText, szBuffer,
              wsprintf (szBuffer, TEXT ("%-20s %7d %7d %7d %7d"), szMapMode,
              rect.left, rect.right, rect.top, rect.bottom)) ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static TCHAR szHeading [] =
       TEXT ("Mapping Mode            Left      Right       Top       Bottom") ;
     static TCHAR szUndLine [] =
       TEXT ("------------            -----     -----       -----     ------") ;
     static int      cxChar, cyChar ;
     HDC             hdc ;
     PAINTSTRUCT     ps ;
     TEXTMETRIC      tm ;
```

```
switch (message)
{
case WM_CREATE:
     hdc = GetDC (hwnd) ;
     SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;

     GetTextMetrics (hdc, &tm) ;
     cxChar = tm.tmAveCharWidth ;
     cyChar = tm.tmHeight + tm.tmExternalLeading ;

     ReleaseDC (hwnd, hdc) ;
     return 0 ;

case WM_PAINT:
     hdc = BeginPaint (hwnd, &ps) ;
     SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
     SetMapMode (hdc, MM_ANISOTROPIC) ;
     SetWindowExtEx (hdc, 1, 1, NULL) ;
     SetViewportExtEx (hdc, cxChar, cyChar, NULL) ;

     TextOut (hdc, 1, 1, szHeading, lstrlen (szHeading)) ;
     TextOut (hdc, 1, 2, szUndLine, lstrlen (szUndLine)) ;

     Show (hwnd, hdc, 1, 3, MM_TEXT,      TEXT ("TEXT (pixels)")) ;
     Show (hwnd, hdc, 1, 4, MM_LOMETRIC,  TEXT ("LOMETRIC (.1 mm)")) ;
     Show (hwnd, hdc, 1, 5, MM_HIMETRIC,  TEXT ("HIMETRIC (.01 mm)")) ;
     Show (hwnd, hdc, 1, 6, MM_LOENGLISH, TEXT ("LOENGLISH (.01 in)")) ;
     Show (hwnd, hdc, 1, 7, MM_HIENGLISH, TEXT ("HIENGLISH (.001 in)")) ;
     Show (hwnd, hdc, 1, 8, MM_TWIPS,     TEXT ("TWIPS (1/1440 in)")) ;

     EndPaint (hwnd, &ps) ;
     return 0 ;

case WM_DESTROY:
     PostQuitMessage (0) ;
     return 0 ;
}
return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

为了便于用 TextOut 函数显示信息，WHATSIZE 使用了一种固定间距的字体。下面一条简单的语句就可以切换为固定间距的字体（在 Windows 3.0 中它是优先使用的）：

```
SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
```

有两个同样的函数用于选取画笔和刷子。像前面提到的，WHATSIZE 也使用 MM_ANIS-

TROPIC 映射方式将逻辑单位设置为字符大小。

当 WHATSIZE 需要获取 6 种映射方式之一的客户区的大小时，它保存当前的设备描述表，设置一种新的映射方式，获取客户区坐标，将它们转换为逻辑坐标，然后在显示信息之前，恢复原映射方式。这些代码在 WHATSIZE 的 Show 函数里：

```
SaveDC (hdc) ;
SetMapMode (hdc, iMapMode) ;
GetClientRect (hwnd, &rect) ;
DptoLP (hdc, (PPOINT) &rect, 2) ;
RestoreDC (hdc, -1) ;
```

图 5-19 显示了 WHATSIZE 的典型输出。

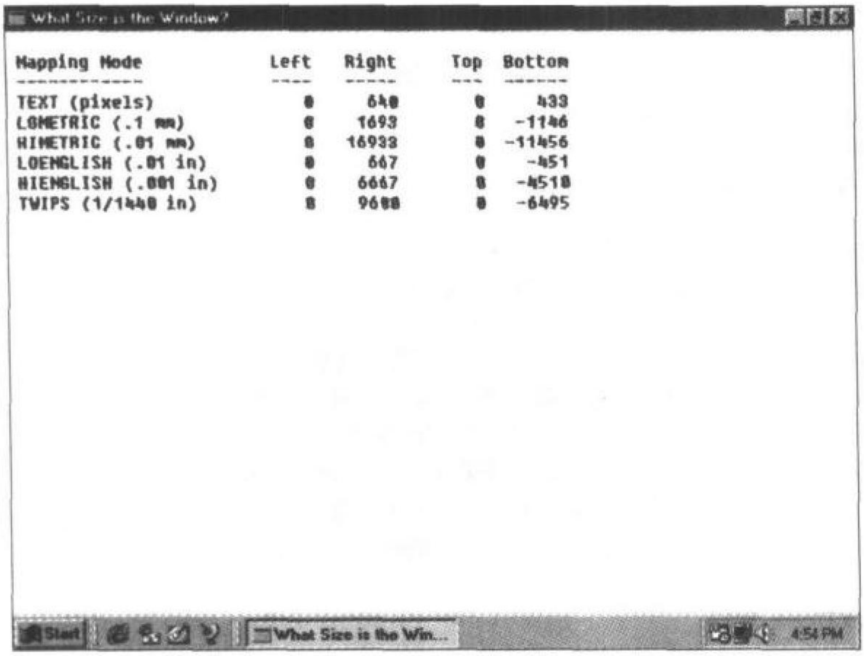


图 5-19 典型的 WHATSIZE 显示

5.6 矩形、区域和剪裁

Windows 包含了几种使用 RECT(矩形)结构和“区域”的绘图函数。区域就是屏幕上的一块地方，它是矩形、多边形和椭圆的组合。

5.6.1 矩形函数

下面三个绘图函数需要一个指向矩形结构的指针：

```
FillRect (hdc, &rect, hBrush) ;
```

```
FrameRect (hdc, &rect, hBrush) ;
InvertRect (hdc, &rect) ;
```

在这些函数中，rect 参数是一个 RECT 类型的结构，它包含有 4 个字段：left、top、right 和 bottom。这个结构中的坐标被当作逻辑坐标。

FillRect 用指定画刷来填充矩形(直到但不包含 right 和 bottom 坐标)，该函数不需要先将画刷选进设备描述表。

FrameRect 使用画刷画矩形框，但是不填充矩形。使用画刷画矩形看起来有点奇怪，因为对于我们所介绍过的函数(如 Rectangle)，其边线都是用当前画笔绘制的。FrameRect 允许用户画一个不一定为纯色的矩形框。该边界框为一个逻辑单位宽。如果逻辑单位大于设备单位，则边界框将会为 2 个像素宽或者更宽。

InvertRect 将矩形中所有像素翻转，1 转换成 0，0 转换为 1，该函数将白色区域转变成黑色，黑色区域转变为白色，绿色区域转变成品红色。

Windows 还提供了 9 个函数，使您可以更容易、更清楚地操作 RECT 结构。例如，要将 RECT 结构的 4 个字段设置为特定值，通常使用如下的程序段：

```
rect.left   = xLeft ;
rect.top    = xTop ;
rect.right  = xRight ;
rect.bottom = xBottom ;
```

但是，通过调用 SetRect 函数，只需要一条语句就可以得到同样的结果：

```
SetRect (&rect, xLeft, yTop, xRight, yBottom) ;
```

在您想要做以下事情之一时，可以很方便地选用其他 8 个函数：

◆ 将矩形沿 x 轴和 y 轴移动几个单元：

```
OffsetRect (&rect, x, y) ;
```

◆ 增减矩形的尺寸：

```
InflateRect (&rect, x, y) ;
```

◆ 矩形各字段设置为 0：

```
SetRectEmpty (&rect) ;
```

◆ 将矩形复制给另一个矩形：

```
CopyRect (&DestRect, &SrcRect) ;
```

◆ 获取两个矩形的交集：

```
IntersectRect (&DestRect, &SrcRect1, &SrcRect2) ;
```

◆ 获取两个矩形的并集：

```
UnionRect (&DestRect, &SrcRect1, &SrcRect2) ;
```

◆ 确定矩形是否为空：

```
bEmpty = IsRectEmpty (&rect) ;
```

◆ 确定点是否在矩形内：

```
bInRect = PtInRect (&rect, point) ;
```

大多数情况下，这些函数的等价代码很简单。例如，您可以用下列语句来替代 CopyRect 函数调用：

```
DestRect = SrcRect ;
```

5.6.2 随机矩形

在图形系统中，有这么一个“永久”运行的有趣程序，它简单地使用随机的大小和色彩绘制一系列矩形。您可以在 Windows 中创建一个这样的程序，但是它并不像乍看起来那样容易编写。我希望您能认识到，您不能简单地在 WM_PAINT 消息中使用一个 while (TRUE)循环。当然，它能够运行，但是程序将停止对其他消息的处理，同时，这个程序不能中止或者最小化。

一种可以接受的方法是设置一个 Windows 计时器，给窗口函数发送 WM_TIMER 消息(我将在第八章中讨论计时器)。对于每条 WM_TIMER 消息，您使用 GetDC 获取一个设备描述表，画一个随机的矩形，然后用 ReleaseDC 释放设备描述表。但是这样又降低了程序的趣味性，因为程序不能尽可能快地画随机矩形，它必须等待 WM_TIMER 消息，而这又依赖于系统时钟的分辨率。

在 Windows 中一定有很多“死时间”，在这个时间内，所有消息队列为空，Windows 只停在一个小循环中等待键盘或者鼠标输入。我们能否在死时间内获得控制，绘制矩形，并且只在有消息加入程序的消息队列之后才释放控制呢？这就是 PeekMessage 函数的目的之一。下面是 PeekMessage 调用的一个例子：

```
PeekMessage (&msg, NULL, 0, 0, PM_REMOVE) ;
```

前面的 4 个参数(一个指向 MSG 结构的指针、一个窗口句柄、两个值指示消息范围)与 GetMessage 的参数相同。将第二、三、四个参数设置为 NULL 或 0 时，表明我们想让 PeekMessage 返回程序中所有窗口的所有消息。如果要将消息从消息队列中删除，则将 PeekMessage 的最后一个参数设置为 PM_REMOVE。如果您不希望删除消息，那么您可以将这个参数设置为 PM_NOREMOVE。这就是为什么 PeekMessage 是“偷看”而不是

"获取"的原因，它使得程序可以检查程序的队列中的下一个消息，而不实际删除它。

GetMessage 不将控制返回给程序，直到从程序的消息队列中获取消息，但是 PeekMessage 总是立刻返回，而不论一个消息是否出现。当消息队列中有一个消息时，PeekMessage 的返回值为 TRUE(非 0)，并且将按通常方式处理消息。当队列中没有消息时，PeekMessage 返回 FALSE(0)。

这使得我们可以改写普通的消息循环。我们可以将如下所示的循环：

```
while (GetMessage (&msg, NULL, 0, 0))
{
     TranslateMessage (&msg) ;
     DispatchMessage (&msg) ;
}
return msg.wParam ;
```

替换为下面的循环：

```
while (TRUE)
{
     if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
     {
          if (msg.message == WM_QUIT)
             break ;

          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     else
     {
          [other program lines to do some work]
     }
}
return msg.wParam ;
```

注意，WM_QUIT 消息是显式检查的。在普通的消息循环中您不必这么做，因为如果 GetMessage 接收到一个 WM_QUIT 消息，它将返回 0，但是 PeekMessage 用它的返回值来指示是否得到一个消息，所以需要对 WM_QUIT 进行检查。

如果 PeekMessage 的返回值为 TRUE，则消息按通常方式进行处理。如果返回值为 FALSE，则在将控制返回给 Windows 之前，还可以做一点工作(如显示另一个随机矩形)。

(尽管 Windows 文档上说，您不能用 PeekMessage 从消息队列中删除 WM_PAINT 消息，但是这并不是什么大不了的问题。毕竟，GetMessage 并不从消息队列中删除 WM_PAINT 消息。从队列中删除 WM_PAINT 消息的唯一方法是令窗口客户区的失效区域变得有效，这可以用 ValidateRect 和 ValidateRgn 或者 BeginPaint 和 EndPaint 对来完成。

如果您在使用 PeekMessage 从队列中取出 WM_PAINT 消息后，同平常一样处理它，那么就不会有问题了。所不能做的是使用如下所示的代码来清除消息队列中的所有消息：

```
while (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE)) ;
```

这条语句从消息队列中删除 WM_PAINT 之外的所有消息。如果队列中有一个 WM_PAINT 消息，程序就会永远地陷在 while 循环中。）

PeekMessage 在 Windows 的早期版本中比在 Windows 98 中要重要得多。这是因为 Windows 的 16 位版本使用的是非抢占式的多任务（我将在第二十章中讨论这一点）。Windows 的 Terminal 程序在从通信线接收输入后，使用一个 PeekMessage 循环。打印管理器程序使用这个技术来进行打印，其他的 Windows 打印应用程序通常都会使用一个 PeekMessage 循环。在 Windows 98 抢占式的多任务环境下，程序可以创建多个线程，我们将第二十章看到这一点。

不管怎样，有了 PeekMessage 函数，我们就可以编写一个不停地显示随机矩形的程序。这个 RANDRECT 如程序 5-7 所示。

程序 5-7　RANDRECT 程序

RANDRECT.C

```
/*----------------------------------------
     RANDRECT.C——Displays Random Rectangles
                 (c) Charles Petzold, 1998
  ----------------------------------------*/

#include <windows.h>
#include <stdlib.h>            // for the rand function

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
void DrawRectangle (HWND) ;

int cxClient, cyClient ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("RandRect") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
```

```
     wndclass.cbWndExtra     = 0 ;
     wndclass.hInstance      = hInstance ;
     wndclass.hIcon          = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground  = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName   = NULL ;
     wndclass.lpszClassName  = szAppName ;
     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Random Rectangles"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (TRUE)
     {
          if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
          {
               if (msg.message == WM_QUIT)
                    break ;
               TranslateMessage (&msg) ;
               DispatchMessage (&msg) ;
          }
          else
               DrawRectangle (hwnd) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
     switch (iMsg)
     {
     case WM_SIZE:
          cxClient = LOWORD (lParam) ;
          cyClient = HIWORD (lParam) ;
          return 0 ;
```

```
    case WM_DESTROY:
        PostQuitMessage (0) ;
        return 0 ;
    }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
void DrawRectangle (HWND hwnd)
{
    HBRUSH  hBrush ;
    HDC     hdc ;
    RECT    rect ;

    if (cxClient == 0 || cyClient == 0)
      return ;

    SetRect (&rect, rand () % cxClient, rand () % cyClient,
                    rand () % cxClient, rand () % cyClient) ;

    hBrush = CreateSolidBrush (
                    RGB (rand () % 256, rand () % 256, rand () % 256)) ;
    hdc = GetDC (hwnd) ;

    FillRect (hdc, &rect, hBrush) ;
    ReleaseDC (hwnd, hdc) ;
    DeleteObject (hBrush) ;
}
```

这个程序在现在的计算机上运行得非常快,看起来都不像是一系列随机矩形了。程序使用我在上面讨论过的 SetRect 和 FillRect 函数,根据由 C 的 rand 函数得到的随机数决定矩形坐标和实心画刷的色彩。我将在第二十章中提供这个程序的多线程版本。

5.6.3 创建和绘制区域

区域是对显示器上一个范围的描述,这个范围是矩形、多边形和椭圆的组合。区域可以用于绘制和剪裁,通过将区域选进设备描述表,就可以用区域来进行剪裁(就是说,将可以绘图的范围限制为客户区的一部分)。与画笔、画刷和位图一样,区域是 GDI 对象,您应该调用 DeleteObject 来删除您所创建的区域。

当您创建一个区域时,Windows 返回一个该区域的句柄,类型为 HRGN。最简单的区域是矩形,有两种创建矩形的方法:

```
hRgn = CreateRectRgn (xLeft, yTop, xRight, yBottom) ;
```

或者

```
hRgn = CreateRectRgnIndirect (&rect) ;
```

您也可以创建椭圆区域：

```
hRgn = CreateEllipticRgn (xLeft, yTop, xRight, yBottom) ;
```

或者

```
hRgn = CreateEllipticRgnIndirect (&rect) ;
```

CreateRoundRectRgn 创建圆角的矩形区域。

创建多边形区域的函数类似于 Polygon 函数：

```
hRgn = CreatePolygonRgn (&point, iCount, iPolyFillMode) ;
```

point 参数是一个 POINT 类型的结构数组，iCount 是点的数目，iPolyFillMode 是 ALTERNATE 或者 WINDING。您还可以用 CreatePolyPolygonRgn 来创建多个多边形区域。

那么，您会问，区域究竟有什么特别之处？下面这个函数才真正显示出区域的作用：

```
iRgnType = CombineRgn (hDestRgn, hSrcRgn1, hSrcRgn2, iCombine) ;
```

这一函数将两个源区域(hSrcRgn1 和 hSrcRgn2)组合起来并用句柄 hDestRgn 指向组合成的目标区域。这三个区域句柄都必须是有效的，但是 hDestRgn 所指向的原来的区域被破坏掉了(当您使用这个函数时，您可能要让 hDestRgn 在初始时指向一个小的矩形区域)。

iCombine 参数说明 hSrcRgn1 和 hSrcRgn2 如何组合，见表 5-9。

表 5-9

iCombine 值	新区域
RGN_AND	两个源区域的公共部分
RGN_OR	两个源区域的全部
RGN_XOR	两个源区域的全部除去公共部分
RGN_DIFF	hSrcRgn1 不在 hSrcRgn2 中的部分
RGN_COPY	hSrcRgn1 的全部(忽略 hSrcRgn2)

从 CombineRgn 返回的 iRgnType 值是下列之一：NULLREGION，表示得到一个空区域；SIMPLEREGION，表示得到一个简单的矩形、椭圆或者多边形；COMPLEXREGION，表示多个矩形、椭圆或多边形的组合；ERROR，表示出错了。

区域的句柄可以用于 4 个绘图函数：

```
FillRgn (hdc, hRgn, hBrush) ;
FrameRgn (hdc, hRgn, hBrush, xFrame, yFrame) ;
InvertRgn (hdc, hRgn) ;
PaintRgn (hdc, hRgn) ;
```

FillRgn、FrameRgn 和 InvertRgn 类似于 FillRect、FrameRect 和 InvertRect。FrameRgn 的 xFrame 和 yFrame 参数是画在区域周围的边框的宽度和高度。PaintRgn 函数用设备描述表中当前画刷填充所指定的区域。所有这些函数都假定区域是用逻辑坐标定义的。

在您用完一个区域后，可以像删除其他 GDI 对象那样删除它：

```
DeleteObject (hRgn) ;
```

5.6.4 矩形与区域的剪裁

区域也在剪裁中扮演了一个角色。InvalidateRect 函数使显示的一个矩形区域失效，并产生一个 WM_PAINT 消息。例如，您可以使用 InvalidateRect 函数来清除客户区，并产生一个 WM_PAINT 消息：

```
InvalidateRect (hwnd, NULL, TRUE) ;
```

您可以通过调用 GetUpdateRect 来获取失效矩形的坐标，并且可以使用 ValidateRect 函数使客户区的矩形有效。当您接收到一个 WM_PAINT 消息时，无效矩形的坐标可以从 PAINTSTRUCT 结构中得到，该结构是用 BeginPaint 函数填充的。这个无效矩形还定义了一个"剪裁区域"，您不能在剪裁区域外绘图。

Windows 有两个作用于区域而不是矩形的函数，它们类似于 InvalidateRect 和 ValidateRect：

```
InvalidateRgn (hwnd, hRgn, bErase) ;
```

和

```
ValidateRgn (hwnd, hRgn) ;
```

当您接收到一个由无效区域引起的 WM_PAINT 消息时，剪裁区域不一定是矩形。

您可以使用以下两个函数之一：

```
SelectObject (hdc, hRgn) ;
```

或

```
SelectClipRgn (hdc, hRgn) ;
```

通过将一个区域选进设备描述表来创建自己的剪裁区域，这个剪裁区域使用设备坐标。

GDI 为剪裁区域建立一份副本，所以在将它选进设备描述表之后，用户可以删除它。Windows 还提供了几个对剪裁区域进行操作的函数，如 ExcludeClipRect 用于将一个矩形从剪裁区域里排除掉，IntersectClipRect 用于创建一个新的剪裁区域，它是前一个剪裁区域与一个矩形的交集，OffsetClipRgn 用于将剪裁区域移动到客户区的另一部分。

5.6.5 CLOVER 程序

CLOVER 程序用 4 个椭圆组成一个区域,将这个区域选进设备描述表中,然后画出从窗口客户区的中心出发的一系列直线,这些直线只出现在区域所限定的范围,结果显示如图 5-20 所示。

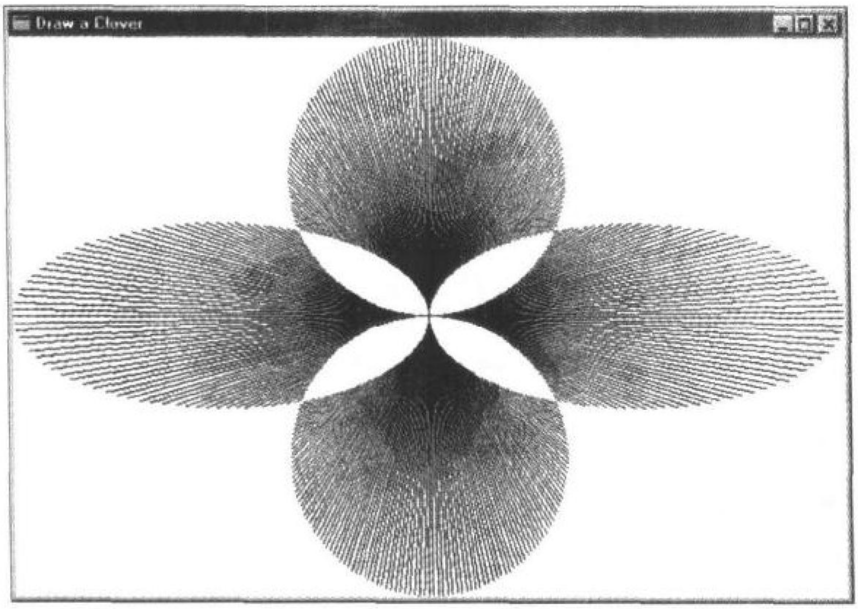


图 5-20 CLOVER 利用复杂的剪裁区域画出的图像

要用常规的方法画出这个图形,就必须根据椭圆的边线公式计算出每条直线的端点。利用复杂的剪裁区域,可以直接画出这些线条,而让 Windows 确定其端点。CLOVER 如程序 5-8 所示。

程序 5-8 CLOVER 程序

CLOVER.C

```
/*-------------------------------------------------------
    CLOVER.C——Clover Drawing Program Using Regions
              (c) Charles Petzold, 1998
  -------------------------------------------------------*/

#include <windows.h>
#include <math.h>

#define TWO_PI (2.0 * 3.14159)
```

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("Clover") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Draw a Clover"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
```

```
{
     static HRGN    hRgnClip ;
     static int     cxClient, cyClient ;
     double         fAngle, fRadius ;
     HCURSOR        hCursor ;
     HDC            hdc ;
     HRGN           hRgnTemp[6] ;
     int            i ;
     PAINTSTRUCT    ps ;

     switch (iMsg)
          {
          case WM_SIZE:
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;
               hCursor = SetCursor (LoadCursor (NULL, IDC_WAIT)) ;
               ShowCursor (TRUE) ;

               if (hRgnClip)
                    DeleteObject (hRgnClip) ;

               hRgnTemp[0] = CreateEllipticRgn (0, cyClient / 3,
                                        cxClient / 2, 2 * cyClient / 3) ;
               hRgnTemp[1] = CreateEllipticRgn (cxClient / 2, cyClient / 3,
                                        cxClient, 2 * cyClient / 3) ;
               hRgnTemp[2] = CreateEllipticRgn (cxClient / 3, 0,
                                        2 * cxClient / 3, cyClient / 2) ;
               hRgnTemp[3] = CreateEllipticRgn (cxClient / 3, cyClient / 2,
                                        2 * cxClient / 3, cyClient) ;
               hRgnTemp[4] = CreateRectRgn (0, 0, 1, 1) ;
               hRgnTemp[5] = CreateRectRgn (0, 0, 1, 1) ;
               hRgnClip    = CreateRectRgn (0, 0, 1, 1) ;

               CombineRgn (hRgnTemp[4], hRgnTemp[0], hRgnTemp[1], RGN_OR) ;
               CombineRgn (hRgnTemp[5], hRgnTemp[2], hRgnTemp[3], RGN_OR) ;
               CombineRgn (hRgnClip,    hRgnTemp[4], hRgnTemp[5], RGN_XOR) ;

               for (i = 0 ; i < 6 ; i++)
                    DeleteObject (hRgnTemp[i]) ;

               SetCursor (hCursor) ;
               ShowCursor (FALSE) ;
               return 0 ;

          case WM_PAINT:
               hdc = BeginPaint (hwnd, &ps) ;

               SetViewportOrgEx (hdc, cxClient / 2, cyClient / 2, NULL) ;
```

```
          SelectClipRgn (hdc, hRgnClip) ;

          fRadius = _hypot (cxClient / 2.0, cyClient / 2.0) ;

          for (fAngle = 0.0 ; fAngle < TWO_PI ; fAngle += TWO_PI / 360)
          {
               MoveToEx (hdc, 0, 0, NULL) ;
               LineTo (hdc, (int) (  fRadius * cos (fAngle) + 0.5),
                            (int) (-fRadius * sin (fAngle) + 0.5)) ;
          }
          EndPaint (hwnd, &ps) ;
          return 0 ;
     case WM_DESTROY:
          DeleteObject (hRgnClip) ;
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

由于区域总是使用设备坐标,CLOVER 程序必须在每次接收到 WM_SIZE 消息时重新创建区域。几年前,这可能需要几秒钟。现在的快速机器在一瞬间就可以画出来。

CLOVER 从创建 4 个椭圆区域开始,这 4 个椭圆存放在 hRgnTemp 数组的头 4 个元素中,然后创建 3 个"哑"区域:

```
hRgnTemp [4] = CreateRectRgn (0, 0, 1, 1) ;
hRgnTemp [5] = CreateRectRgn (0, 0, 1, 1) ;
hRgnClip     = CreateRectRgn (0, 0, 1, 1) ;
```

客户区左右的两个椭圆区域组合起来:

```
CombineRgn (hRgnTemp [4], hRgnTemp [0], hRgnTemp [1], RGN_OR) ;
```

同样,客户区上下两个椭圆区域组合起来:

```
CombineRgn (hRgnTemp [5], hRgnTemp [2], hRgnTemp [3], RGN_OR) ;
```

最后,两个组合后的区域再组合到 hRgnClip 中:

```
CombineRgn (hRgnClip, hRgnTemp [4], hRgnTemp [5], RGN_XOR) ;
```

RGN_XOR 标识符用于从结果区域中排除重叠部分。最后,删除 6 个临时区域:

```
for (i = 0 ; i < 6 ; i++)
     DeleteObject (hRgnTemp [i]) ;
```

与画出的图形比起来,WM_PAINT 的处理很简单。视口原点设置为客户区的中心(使画直线更容易一些),在 WM_SIZE 消息期间创建的区域选择为设备描述表的剪裁

区域：

```
SetViewportOrg (hdc, xClient / 2, yClient / 2) ;
SelectClipRgn (hdc, hRgnClip) ;
```

现在，剩下的就是画直线了，共360条，每隔一度画一条。每条线的长度为变量fRadius，这是从中心到客户区的角落的距离：

```
fRadius = hypot (xClient / 2.0, yClient / 2.0) ;

for (fAngle = 0.0 ; fAngle < TWO_PI ; fAngle += TWO_PI / 360)
{
     MoveToEx (hdc, 0, 0, NULL) ;
     LineTo (hdc, (int) (  fRadius * cos (fAngle) + 0.5),
                  (int) (- fRadius * sin (fAngle) + 0.5)) ;
}
```

在处理WM_DESTROY消息时，删除该区域：

```
DeleteObject (hRgnClip) ;
```

这不是本书图形编程的最后内容。我将在第十三章讨论打印，第十四章和十五章讨论位图，第十七章讨论文本和字体，第十八章讨论元文件。

第六章

键盘

在 Microsoft Windows 98 中，键盘和鼠标是两个标准的用户输入源，在一些交叠的操作中通常相互补充使用。当然，鼠标在今天的应用程序中比 10 年前使用得更为广泛。甚至在一些应用程序中，我们更习惯于使用鼠标，例如在游戏、画图程序、音乐程序，以及 Web 浏览器等程序中就是这样。然而，我们可以不使用鼠标，但绝对不能从一般的 PC 中拆掉键盘。

相对于个人计算机的其他组件，键盘有非常久远的历史，它起源于 1874 年的第一台 Remington 打字机。早期的计算机程序员用键盘在 Hollerith 卡片上打孔，以后在哑终端上用键盘直接与大型主机通讯。PC 上的键盘在某些方面进行了扩展，包括了功能键、光标定位键和（通常都带有的）单独的数字键盘，但它们的输入原理基本相同。

6.1 键盘基础

Windows 程序获得键盘输入的方式：键盘输入以消息的形式传递给程序的窗口过程。实际上，第一次学习消息时，键盘就是一个明显的例子：消息应该传递给应用程序的信息类型。

Windows 用 8 种不同的消息来传递不同的键盘事件。这好像太多了，但是（就像我们所看到的一样）程序可以忽略其中至少一半的消息而不会有任何问题。并且，在大多数情况下，这些消息中包含的键盘信息会多于程序所需要的。处理键盘的部分工作就是识别出哪些消息是重要的，哪些是不重要的。

6.1.1 忽略键盘

虽然键盘是 Windows 程序中用户输入的主要来源，但是程序不必对它接收的所有消息都做出响应。Windows 本身也能处理许多键盘功能。

例如，您可以忽略那些属于系统功能的击键，它们通常用到 Alt 键。程序不必监视这些击键，因为 Windows 会将击键的作用通知程序（当然，如果程序想这么做，它也能监视这些击键）。虽然调用程序菜单的击键将通过窗口的窗口过程，但通常默认的处理方式是将击键传递给 DefWindowProc。最终，窗口过程将获得一个消息，表明选择了一条菜单项。通常，这是所有窗口过程需要知道的。（在第十章将介绍菜单。）

很多 Windows 程序使用“键盘加速键”来激活通用菜单项。加速键通常是功能键或字母同 Ctrl 键的组合（例如，Ctrl-S 用于保存文件）。这些键盘加速键与程序菜单一起在程序的资源描述文件中定义（我们可以在第十章看到）。Windows 将这些键盘加速键转换为菜单命令消息，您不必自己去进行转换。

对话框也有键盘接口，但是当对话框处于活动状态时，应用程序通常不必监视键盘。键盘接口由 Windows 处理，Windows 把关于击键作用的消息发送给程序。对话框可以包含用于输入文本的编辑控制。它们一般是小方框，用户可以在框中键入字符串。Windows 处理所有编辑控制逻辑，并在输入完毕后，将编辑控制的最终内容传送给程序。关于对话框的详细信息，请参见第十一章。

编辑控制不必局限于单独一行，而且也不限于只在对话框中。程序主窗口的一个多行编辑控制能够作为一个基本的文本编辑器。（请参见第九、十、十一和十三章的 POPPAD 程序。）Windows 甚至有一个多信息文本编辑控制，允许您编辑和显示格式化的文本。（请参见/Platform SDK/User Interface Services/Controls/Rich Edit Controls。）

您将会发现，在构造 Windows 程序时，能够使用处理键盘和鼠标输入的子窗口控制来将较高层的信息传递回父窗口。使用足够多的此类控制，您就不会因处理键盘消息而烦恼。

6.1.2 谁获得了焦点

与所有的个人计算机硬件一样，键盘必须由在 Windows 下运行的所有应用程序共享。有些应用程序可能有多个窗口，键盘必须由该应用程序内的所有窗口共享。

回想一下，程序用来从消息队列中检索消息的 MSG 结构包括 hwnd 字段。此字段指出接收消息的窗口句柄。消息循环中的 DispatchMessage 函数向窗口过程发送该消息，此窗口过程与需要消息的窗口相联系。当按下键盘上的键时，只有一个窗口过程接收键盘消息，并且此消息包括接收消息的窗口句柄。

接收特定键盘事件的窗口具有输入焦点。输入焦点的概念与活动窗口的概念很相近。有输入焦点的窗口是活动窗口或活动窗口的派生窗口（活动窗口的子窗口，或者活动窗口子窗口的子窗口，等等）。

通常很容易辨别活动窗口。它通常是顶级窗口——也就是说，它的父窗口句柄是 NULL。如果活动窗口有标题栏，Windows 将突出显示标题栏。如果活动窗口具有对话

框架(对话框中很常见的格式)而不是标题栏,Windows 将突出显示框架。如果活动窗口当前是最小化的,那么 Windows 将在任务栏中突出显示该项,其显示就像一个按下的按钮。

如果活动窗口有子窗口,那么有输入焦点的窗口既可以是活动窗口也可以是其子窗口。最常见的子窗口类似以下控制:出现在对话框中的下压按钮、单选钮、复选框、滚动条、编辑框和列表框。子窗口不能自己成为活动窗口。只有当它是活动窗口的派生窗口时,子窗口才能有输入焦点。子窗口控制一般通过显示一个闪烁的插入符或虚线来表明它具有输入焦点。

有时没有窗口具有输入焦点。这种情况发生在所有程序都是最小化的时候。这时,Windows 将继续向活动窗口发送键盘消息,但是这些消息与发送给非最小化的活动窗口的键盘消息有不同的形式。

窗口过程通过捕获 WM_SETFOCUS 和 WM_KILLFOCUS 消息来判定它的窗口何时拥有输入焦点。WM_SETFOCUS 指示窗口正在得到输入焦点,WM_KILLFOCUS 表示窗口正在失去输入焦点。我将在本章的后面详细说明这些消息。

6.1.3 队列和同步

当用户按下并释放键盘上的键时,Windows 和键盘驱动程序将硬件扫描码转换为格式消息。然而,这些消息并不保存在消息队列中。实际上,Windows 在所谓的“系统消息队列”中保存这些消息。系统消息队列是单消息队列,它由 Windows 维护,用于初步保存用户从键盘和鼠标输入的信息。只有当 Windows 应用程序处理完前一个用户输入消息时,Windows 才会从系统消息队列中取出下一个消息,并将其放入应用程序的消息队列中。

此过程分为两步:首先在系统消息队列中保存消息,然后将它们放入应用程序的消息队列,其原因是需要同步。就像我们刚才所学的,假定接收键盘输入的窗口就是有输入焦点的窗口。用户的输入速度可能比应用程序处理击键的速度快,并且特定的击键可能会使焦点从一个窗口切换到另一个窗口,后来的击键就输入到了另一个窗口。但如果后来的击键已经记下了目标窗口的地址,并放入了应用程序消息队列,那么后来的击键就不能输入到另一个窗口。

6.1.4 击键和字符

应用程序从 Windows 接收的关于键盘事件的消息可以分为击键和字符两类,这同您看待键盘的两种方式一致。

首先,您可以将键盘看作是键的集合。键盘只有唯一的 A 键,按下该键是一次击键,释放该键也是一次击键。但是键盘也是能产生可显示字符或控制字符的输入设备。

根据 Ctrl、Shift 和 Caps Lock 键的状态，A 键能产生几个字符。通常情况下，此字符为小写 a。如果按下 Shift 键或者打开了 Caps Lock，则该字符就变成大写 A。如果按下了 Ctrl，则该字符为 Ctrl-A(它在 ASCII 中有意义，但在 Windows 中可能是某事件的键盘加速键)。在一些键盘上，A 击键之前可能有"死字符键(dead-character key)"或者 Shift、Ctrl 或者 Alt 的不同组合，这些组合可以产生带有音调标记的小写或者大写，例如，à、á、â、ã、Ä 或 Å。

对产生可显示字符的击键组合，Windows 不仅给程序发送击键消息，而且还发送字符消息。有些键不产生字符，这些键包括 shift 键、功能键、光标移动键和特殊字符键如 Insert 和 Delete。对于这些键，Windows 只产生击键消息。

6.2 击键消息

当您按下一个键时，Windows 把 WM_KEYDOWN 或者 WM_SYSKEYDOWN 消息放入有输入焦点的窗口的消息队列；当您释放一个键时，Windows 把 WM_KEYUP 或者 WM_SYSKEYUP 消息放入消息队列中(见表 6-1)。

表 6-1

	键按下	键释放
非系统键	WM_KEYDOWN	WM_KEYUP
系统键	WM_SYSKEYDOWN	WM_SYSKEYUP

通常"down(按下)"和"up(释放)"消息是成对出现的。不过，如果您按住一个键使得自动重复功能生效，那么当该键最后被释放时，Windows 会给窗口过程发送一系列 WM_KEYDOWN(或者 WM_SYSKEYDOWN)消息和一个 WM_KEYUP(或者 WM_SYSKEYUP)消息。像所有入队列的消息一样，击键消息也有时间信息。通过调用 GetMessageTime，您可以获得按下或者释放键的相对时间。

6.2.1 系统击键与非系统击键

WM_SYSKEYDOWN 和 WM_SYSKEYUP 中的"SYS"代表"系统"，它表示该击键对 Windows 比对 Windows 应用程序更加重要。WM_SYSKEYDOWN 和 WM_SYSKEYUP 消息经常由与 Alt 相组合的击键产生，这些击键激活程序菜单或者系统菜单上的选项，或者用于切换活动窗口等系统功能(Alt-Tab 或者 Alt-Esc)，也可以用作系统菜单加速键(Alt 键与一个功能键相结合，例如 Alt-F4 用于关闭应用程序)。程序通常忽略 WM_SYSKEYUP 和 WM_SYSKEYDOWN 消息，并将它们传送到 DefWindowProc。由于 Windows 要处理所有 Alt 键的功能，所以您无需捕获这些消息。您的窗口过程将最后收到关于这些击键

结果(如菜单选择)的其他消息。如果您想在自己的窗口过程中包括捕获系统击键的代码(如本章后面的 KEYVIEW1 和 KEYVIEW2 程序所做的那样),那么在处理这些消息之后再传送到 DefWindowProc,Windows 就仍然可以将它们用于通常的目的。

但是,请再考虑一下,几乎所有会影响用户程序窗口的消息都会先通过用户窗口过程。只有用户把消息传送到 DefWindowProc,Windows 才会对消息进行处理。例如,如果您将语句

```
case WM_SYSKEYDOWN:
case WM_SYSKEYUP:
case WM_SYSCHAR:
    return 0 ;
```

加入到一个窗口过程中,那么当您的程序主窗口拥有输入焦点时,就可以有效地禁用所有 Alt 键操作(我将在本章的后面讨论 WM_SYSCHAR),其中包括 Alt-Tab、Alt-Esc,以及菜单操作。虽然我怀疑您会这么做,但是,我相信您会感到窗口过程的强大功能。

WM_KEYDOWN 和 WM_KEYUP 消息通常是在按下或者释放不带 Alt 键的键时产生的,您的程序可以使用或者忽略这些消息,Windows 本身并不处理这些消息。

对所有 4 类击键消息,wParam 是虚拟键代码,表示按下或释放的键,而 lParam 则包含属于击键的其他数据。

6.2.2 虚拟键码

虚拟键码保存在 WM_KEYDOWN、WM_KEYUP、WM_SYSKEYDOWN 和 WM_SYSKEYUP 消息的 wParam 参数中。此代码标识按下或释放的键。

哈,又是"虚拟",您喜欢它吗?虚拟指的是假定存在于思想中而不是现实世界中的一些事物,也只有熟练使用 DOS 汇编语言编写应用程序的程序员才有可能指出,为什么对 Windows 键盘处理如此基本的键码是虚拟的而不是真实的。

对于早期的程序员来说,真实的键码由物理键盘硬件产生。在 Windows 文档中将这些键码称为"扫描码(scan codes)"。在 IBM 兼容机上,扫描码 16 是 Q 键,17 是 W 键,18 是 E, 19 是 R, 20 是 T, 21 是 Y 等等。这时您会发现,扫描码是基于键盘的物理布局的。Windows 开发者认为这些代码过于与设备相关了,于是他们试图通过定义所谓的虚拟键码来以与设备无关的方式处理键盘。其中一些虚拟键码不能在 IBM 兼容机上产生,但可能会在其他制造商生产的键盘中找到,或者在未来的键盘上找到。

您使用的大多数虚拟键码的名称在 WINUSER.H 头文件中都定义为以 VK_开头。表 6-2 列出了这些名称和数值(十进制和十六进制),以及与虚拟键相对应的 IBM 兼容机键盘上的键。下表也标出了 Windows 运行时是否需要这些键。下表还按数字顺序列出了虚拟键码。

前 4 个虚拟键码中有 3 个指的是鼠标键，如表 6-2 所示。您永远都不会从键盘消息中获得这些鼠标键代码。在下一章可以看到，我们能够从鼠标消息中获得它们。VK_CANCEL 代码是一个虚拟键码，它包括同时按下两个键(Ctrl-Break)。Windows 应用程序通常不使用此键。

表 6-2

十进制	十六进制	WINUSER.H 标识符	必需?	IBM 兼容键盘
1	01	VK_LBUTTON		鼠标左键
2	02	VK_RBUTTON		鼠标右键
3	03	VK_CANCEL	√	Ctrl-Break
4	04	VK_MBUTTON		鼠标中键

表 6-3 中的键——Backspace、Tab、Enter、Escape 和 Spacebar——通常用于 Windows 程序。不过，Windows 一般用字符消息(而不是键盘消息)来处理这些键。另外，Windows 程序通常不需要监视 Shift、Ctrl 或 Alt 键的状态。

表 6-3

十进制	十六进制	WINUSER.H 标识符	必需?	IBM 兼容键盘
8	08	VK_BACK	√	Backspace
9	09	VK_TAB	√	Tab
12	0C	VK_CLEAR		Num Lock 关闭时的数字键盘 5
13	0D	VK_RETURN	√	Enter(或者另一个)
16	10	VK_SHIFT	√	Shift(或者另一个)
17	11	VK_CONTROL	√	Ctrl(或者另一个)
18	12	VK_MENU	√	Alt(或者另一个)
19	13	VK_PAUSE		Pause
20	14	VK_CAPITAL	√	Caps Lock
27	1B	VK_ESCAPE	√	Esc
32	20	VK_SPACE	√	Spacebar

表 6-4 列出的前 8 个码可能是与 VK_INSERT 和 VK_DELETE 一起最常用的虚拟键码。

注意，许多名称(例如 VK_PRIOR 和 VK_NEXT)都与键上的标志不同，而且也与滚动条中的标识符不统一。Print Screen 键在很大程度上被 Windows 应用程序所忽略。Windows 本身通过将视频显示的副本按位图保存到剪贴板来响应此键。VK_SELECT、

VK_PRINT、VK_EXECUTE 和 VK_HELP 将在假定的、我们很少见的键盘上出现。

表 6-4

十进制	十六进制	WINUSER.H 标识符	必需?	IBM 兼容键盘
33	21	VK_PRIOR	√	Page Up
34	22	VK_NEXT	√	Page Down
35	23	VK_END	√	End
36	24	VK_HOME	√	Home
37	25	VK_LEFT	√	左箭头
38	26	VK_UP	√	上箭头
39	27	VK_RIGHT	√	右箭头
40	28	VK_DOWN	√	下箭头
41	29	VK_SELECT		
42	2A	VK_PRINT		
43	2B	VK_EXECUTE		
44	2C	VK_SNAPSHOT		Print Screen
45	2D	VK_INSERT	√	Insert
46	2E	VK_DELETE	√	Delete
47	2F	VK_HELP		

Windows 也包括主键盘上的字母和数字键的虚拟键码(见表 6-5)。(数字键盘将单独处理。)

表 6-5

十进制	十六进制	WINUSER.H 标识符	必需?	IBM 兼容键盘
48 ~ 57	30 ~ 39	无	√	主键盘上的 0 ~ 9
65 ~ 90	41 ~ 5A	无	√	A ~ Z

注意,数字和字母的虚拟键码是 ASCII 码。Windows 程序几乎从不使用这些虚拟键码;实际上,程序依赖于 ASCII 码字符的字符消息。

Windows 用 VK_LWIN 和 VK_RWIN 键打开"开始"菜单或者(在以前的版本中)启动"任务管理器"。这两个都可以用于登录或注销 Windows(只在 Microsoft Windows NT 中有效),或者登录或注销网络(在面向工作组的 Windows 中)。应用程序能够通过显示帮助

信息或者快捷方式来处理 application 键。表 6-6 所示的代码是由 Microsoft Natural Keyboard 及其兼容键盘产生的。

表 6-6

十进制	十六进制	WINUSER.H 标识符	必需?	Microsoft 自然键盘
91	5B	VK_LWIN		左 Windows 键
92	5C	VK_RWIN		右 Windows 键
93	5D	VK_APPS		Applications 键

表 6-7 所示的代码用于数字键盘上的键(如果有的话)。

表 6-7

十进制	十六进制	WINUSER.H 标识符	必需?	IBM 兼容键盘
96 ~ 105	60 ~ 69	VK_NUMPAD0 ~ VK_NUMPAD9		Num Lock 打开时数字键盘上的 0 ~ 9
106	6A	VK_MULTIPLY		数字键盘上的 *
107	6B	VK_ADD		数字键盘上的 +
108	6C	VK_SEPARATOR		
109	6D	VK_SUBTRACT		数字键盘上的 -
110	6E	VK_DECIMAL		数字键盘上的 .
111	6F	VK_DIVIDE		数字键盘上的/

最后,虽然多数的键盘都有 12 个功能键,但 Windows 只需要 10 个,而数字标识却有 24 个。另外,程序通常用功能键作为键盘加速键,这样,它们通常不处理表 6-8 所示的击键。

表 6-8

十进制	十六进制	WINUSER.H 标识符	必需?	IBM 兼容键盘
112 ~ 121	70 ~ 79	VK_F1 ~ VK_F10	√	功能键 F1 ~ F10
122 ~ 135	7A ~ 87	VK_F11 ~ VK_F24		功能键 F11 ~ F24
144	90	VK_NUMLOCK		Num Lock
145	91	VK_SCROLL		Scroll Lock

另外,还定义了一些其他虚拟键码,但它们只用于非标准键盘上的键,或者通常在大型机终端上使用的键。请查看/Platform SDK/User Interface Services/User Input/Virtual-

Key Codes,可得到完整的列表。

6.2.3 lParam 信息

在 4 个击键消息(WM_KEYDOWN、WM_KEYUP、WM_SYSKEYDOWN 和 WM_SYSKEYUP)中,wParam 消息参数含有上面所讨论的虚拟键码,而 lParam 消息参数则含有对了解击键非常有用的其他信息。lParam 的 32 位分为 6 个域,如图 6-1 所示。

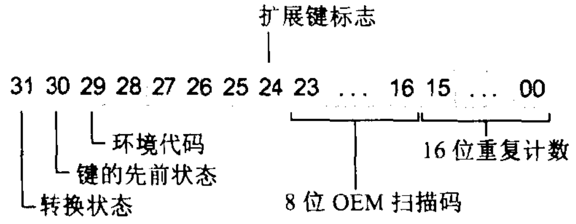


图 6-1 lParam 变量的 6 个击键消息域

重复计数

重复计数是该消息所表示的击键次数,大多数情况下,重复计数设置为 1。不过,如果按下一个键之后,您的窗口过程不够快,以致不能处理自动重复速率(您可以在"控制面板"的"键盘"中进行设置)下的按键消息,Windows 就把几个 WM_KEYDOWN 或者 WM_SYSKEYDOWN 消息组合到单个消息中,并相应地增加重复计数。WM_KEYUP 或 WM_SYSKEYUP 消息的重复计数总为 1。

因为重复计数大于 1 指示出击键速率大于您程序的处理能力,所以您也可能想在处理键盘消息时忽略重复计数。几乎每个人都有字处理文档或电子表格滚动过快的经历,因为多余的击键堆满了键盘缓冲区,所以当程序用一些时间来处理每一次击键时,如果忽略您程序中的重复计数,就能够解决此问题。不过,有时可能也会用到重复计数,您应该尝试使用两种方法运行程序,从中找出一种较好的方法。

OEM 扫描码

OEM 扫描码是由硬件(键盘)产生的代码。这对于中年的汇编语言程序员来说很熟悉,它是从 PC 兼容机的 ROM BIOS 服务中获得的值。(OEM 指的是 PC 的原始设备制造商(Original Equipment Manufacturer),在本文中是"IBM 标准"同义语。)我们现在不再需要这些信息了。Windows 程序能够忽略几乎所有的 OEM 扫描码,除非它取决于键盘的物理布局,请参见第二十二章的程序 KBMIDI。

扩展键标志

如果击键结果来自 IBM 增强键盘的附加键之一，那么扩展键标志为 1(IBM 增强型键盘有 101 或 102 个键。功能键在顶端。光标移动键从数字键盘中分离出来，但在数字键盘上还保留有光标移动键的副本)。对键盘右端的 Alt 和 Ctrl 键，以及不是数字键盘那部分的光标移动键(包括 Insert 和 Delete 键)、数字键盘上的斜杠(/)和 Enter 键，以及 Num Lock 键等，此标志均被设置为 1。Windows 程序通常忽略扩展键标志。

环境代码

环境代码在按下 Alt 键后为 1。对 WM_SYSKEYUP 和 WM_SYSKEYDOWN 消息，这一位总是 1；对 WM_KEYUP 和 WM_KEYDOWN 消息，这一位总是 0；但是有两个例外：

- 如果活动窗口最小化了，则它没有输入焦点。这时候所有的击键都产生 WM_SYSKEYUP 和 WM_SYSKEYDOWN 消息。如果 Alt 键未被按下，则环境代码域被设置为 0。Windows 使用 WM_SYSKEYUP 和 WM_SYSKEYDOWN 消息，从而使最小化了的活动窗口不处理这些击键。
- 对于一些外国语键盘，有些字符是通过 Shift、Ctrl 或者 Alt 键与其他键相组合产生的。这时环境代码为 1，但是此消息并非系统击键消息。

键的先前状态

如果在此之前键是释放的，则键的先前状态为 0，否则为 1。对 WM_KEYUP 或者 WM_SYSKEYUP 消息，它总是设置为 1；但是对 WM_KEYDOWN 或者 WM_SYSKEYDOWN 消息，此位可以为 0，也可以为 1。如果为 1，则表示该键是自动重复功能所产生的第二个或者后续消息。

转换状态

如果键正被按下，则转换状态为 0；如果键正被释放，则转换状态为 1。对 WM_KEYDOWN 或者 WM_SYSKEYDOWN 消息，此域为 0；对 WM_KEYUP 或者 WM_SYSKEYUP 消息，此域为 1。

6.2.4 换档状态

在处理击键消息时，您可能需要知道是否按下了换档键(Shift、Ctrl 和 Alt)或开关键(Caps Lock、Num Lock 和 Scroll Lock)。通过调用 GetKeyState 函数，您就能获得此信息。例如：

```
iState = GetKeyState (VK_SHIFT) ;
```

如果按下了 Shift,则 iState 值为负(即高位被置位)。如果 Caps Lock 键打开,则从

```
iState = GetKeyState (VK_CAPITAL) ;
```

返回的值低位被置位。此位与键盘上的小灯保持一致。

通常,您在使用 GetKeyState 时,会带有虚拟键码 VK_SHIFT、VK_CONTROL 和 VK_MENU(在说明 Alt 键时调用)。使用 GetKeyState 时,您也可以用下面的标识符来确定按下的 Shift、Ctrl 或 Alt 键是左边的还是右边的:VK_LSHIFT、VK_RSHIFT、VK_LCONTROL、VK_RCONTROL、VK_LMENU 和 VK_RMENU。这些标识符只用于 GetKeyState 和 GetAsyncKeyState(下面将详细说明)。

使用虚拟键码 VK_LBUTTON、VK_RBUTTON 和 VK_MBUTTON,您也可以获得鼠标键的状态。不过,大多数需要监视鼠标键与击键相组合的 Windows 应用程序都使用其他方法来做到这一点——即在接收到鼠标消息时检查击键。实际上,换档状态信息包含在鼠标信息中,正如您在下一章中将看到的一样。

请注意 GetKeyState 的使用,它并非实时检查键盘状态,而只是检查当前正在处理的消息发生之前和发生之时的键盘状态。多数情况下,这就是您所需要的。如果您需要确定用户是否按下了 Shift-Tab,请在处理 Tab 键的 WM_KEYDOWN 消息时调用带有参数 VK_SHIFT 的 GetKeyState。如果 GetKeyState 的返回值为负,那么您就知道在按下 Tab 键之前用户按下了 Shift 键。并且,如果在您开始处理 Tab 键之前,已经释放了 Shift 键也没有关系。您知道,在按下 Tab 键的时候 Shift 键是按下的。

GetKeyState 不会让您获得独立于普通键盘消息的键盘信息。例如,您或许想暂停窗口过程的处理,直到您按下 F1 功能键为止:

```
while (GetKeyState (VK_F1) >= 0) ;     // WRONG !!!
```

不要这么做!这将挂起程序(当然,除非在执行此语句之前就从消息队列中接收到了 F1 的 WM_KEYDOWN)。如果您确实需要知道当前某键的状态,那么您可以使用 GetAsyncKeyState。

6.2.5 使用击键消息

如果程序能够获得关于每个击键的信息,这当然很理想,但是大多数 Windows 程序忽略了几乎所有的击键,而只处理几条击键消息。WM_SYSKEYDOWN 和 WM_SYSKEYUP 消息是由 Windows 系统函数使用的,您不必为此费心,就算您要处理 WM_KEYDOWN 消息,通常也可以忽略 WM_KEYUP 消息。

Windows 程序通常为不产生字符的击键使用 WM_KEYDOWN 消息。虽然您可以认

为借助击键消息和换档键状态信息能将击键消息转换为字符消息，但是不要这么做，因为您将遇到国际键盘间的不同所带来的问题。例如，如果您得到 wParam 等于 0x33 的 WM_KEYDOWN 消息，您就可以知道用户按下了键 3，到此为止一切正常。这时，如果用 GetKeyState 发现 Shift 键被按下，您就可能会认为用户输入了 # 号。这可不一定，比如英国用户就是在输入 £。

对于光标移动键、功能键、Insert 和 Delete 键，WM_KEYDOWN 消息是最有用的。不过，Insert、Delete 和功能键经常作为菜单加速键。因为 Windows 能把菜单加速键翻译为菜单命令消息，所以您就不必自己来处理击键。

在 Windows 之前的 MS-DOS 应用程序中大量使用功能键与 Shift、Ctrl 和 Alt 键的组合，同样地，您也可以在 Windows 程序中使用(实际上，Microsoft Word 将大量的功能键用作命令快捷方式)，但并不推荐这样做。如果您确实希望使用功能键，那么这些键应该是重复菜单命令。Windows 的目标之一就是提供不需要记忆或者使用复杂命令流程的用户界面。

因此，可以归纳如下：多数情况下，您将只处理光标移动键(有时也为 Insert 和 Delete 键)的 WM_KEYDOWN 消息。在使用这些键的时候，您可以通过 GetKeyState 来检查 Shift 键和 Ctrl 键的状态。例如，Windows 程序经常使用 Shift 与光标键的组合键来扩大字处理文档里选中的范围。Ctrl 键常用于修改光标键的意义。例如，Ctrl 与右箭头键相组合可以表明光标右移一个字。

决定程序中使用键盘方式的最佳方法之一是了解现有的 Windows 程序使用键盘的方式。如果您不喜欢那些定义，当然可以对其加以修改，但是这样做不利于其他人很快地学会使用您的程序。

6.2.6 为键盘增强 SYSMETS

当编写第四章中三个版本的 SYSMETS 程序时，我们还不了解键盘，只能使用滚动条和鼠标来滚动文本。现在我们知道了处理键盘消息的方法，那么不妨在程序中加入键盘接口。显然，这是处理光标移动键的工作。我们将大多数光标键(Home、End、Page Up、Page Down、Up Arrow 和 Down Arrow)用于垂直滚动，左箭头键和右箭头键用于不太重要的水平滚动。

创建键盘接口的一种简单方法是在窗口过程中加入与 WM_VSCROLL 和 WM_HSCROLL 逻辑相平行的而且本质上也是其副本的 WM_KEYDOWN 逻辑。不过，这也是不明智的，因为如果要修改滚动条逻辑，那么必须相应地修改 WM_KEYDOWN。

为什么不简单地将每一种 WM_KEYDOWN 消息都翻译成等价的 WM_VSCROLL 或者 WM_HSCROLL 消息呢？通过向窗口过程发送假冒消息，我们可能会让 WndProc 认为它获得了滚动条信息。

在 Windows 中,这种方法是可行的。发消息的函数叫做 SendMessage,它所用的参数与传递到窗口过程的参数是相同的:

```
SendMessage (hwnd, message, wParam, lParam) ;
```

当调用 SendMessage 时,Windows 调用窗口句柄为 hwnd 的窗口过程,并把这 4 个参数传给它。当窗口过程完成消息处理之后,Windows 把控制返回到 SendMessage 调用之后的下一条语句。您将消息发送到的窗口过程可以是同一个窗口过程、同一程序中的其他窗口过程,或甚至是其他应用程序中的窗口过程。

下面说明在 SYSMETS 程序中使用 SendMessage 处理 WM_KEYDOWN 代码的方法:

```
case WM_KEYDOWN:
    switch (wParam)
    {
    case VK_HOME:
        SendMessage (hwnd, WM_VSCROLL, SB_TOP, 0) ;
        break ;

    case VK_END:
        SendMessage (hwnd, WM_VSCROLL, SB_BOTTOM, 0) ;
        break ;

    case VK_PRIOR:
        SendMessage (hwnd, WM_VSCROLL, SB_PAGEUP, 0) ;
        break ;
```

至此您已经有了总的印象。我们的目标是为滚动条添加键盘接口,并且也正在这么做。通过把滚动条消息发送到窗口过程,我们实现了用光标移动键重复滚动条的功能。现在您知道在 SYSMETS3 中为 WM_VSCROLL 消息引入对 SB_TOP 和 SB_BOTTOM 处理的原因了。在那里并没有用到它,但是现在处理 Home 和 End 键时就有用了。如程序 6-1 所示的 SYSENTS4 就体现了这些变化。编译这个程序时还需要第四章的 SYSMETS.H 文件。

程序 6-1 SYSMETS4 程序

SYSMETS4.C

```
/*-------------------------------------------------
    SYSMETS4.C——System Metrics Display Program No. 4
                (c) Charles Petzold, 1998
  -------------------------------------------------*/

#include <windows.h>
#include "sysmets.h"
```

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("SysMets4") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Get System Metrics No. 4"),
                          WS_OVERLAPPEDWINDOW | WS_VSCROLL | WS_HSCROLL,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
```

```
static int    cxChar, cxCaps, cyChar, cxClient, cyClient, iMaxWidth ;
HDC           hdc ;
int           i, x, y, iVertPos, iHorzPos, iPaintBeg, iPaintEnd ;
PAINTSTRUCT   ps ;
SCROLLINFO    si ;
TCHAR         szBuffer[10] ;
TEXTMETRIC    tm ;

switch (message)
{
case WM_CREATE:
    hdc = GetDC (hwnd) ;

    GetTextMetrics (hdc, &tm) ;
    cxChar = tm.tmAveCharWidth ;
    cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
    cyChar = tm.tmHeight + tm.tmExternalLeading ;

    ReleaseDC (hwnd, hdc) ;

         // Save the width of the three columns

    iMaxWidth = 40 * cxChar + 22 * cxCaps ;
    return 0 ;

case WM_SIZE:
    cxClient = LOWORD (lParam) ;
    cyClient = HIWORD (lParam) ;

         // Set vertical scroll bar range and page size

    si.cbSize = sizeof (si) ;
    si.fMask  = SIF_RANGE | SIF_PAGE ;
    si.nMin   = 0 ;
    si.nMax   = NUMLINES - 1 ;
    si.nPage  = cyClient / cyChar ;
    SetScrollInfo (hwnd, SB_VERT, &si, TRUE) ;

         // Set horizontal scroll bar range and page size

    si.cbSize = sizeof (si) ;
    si.fMask  = SIF_RANGE | SIF_PAGE ;
    si.nMin   = 0 ;
    si.nMax   = 2 + iMaxWidth / cxChar ;
    si.nPage  = cxClient / cxChar ;
    SetScrollInfo (hwnd, SB_HORZ, &si, TRUE) ;
    return 0 ;
```

```
case WM_VSCROLL:
        // Get all the vertical scroll bar information

    si.cbSize = sizeof (si) ;
    si.fMask  = SIF_ALL ;
    GetScrollInfo (hwnd, SB_VERT, &si) ;

        // Save the position for comparison later on

    iVertPos = si.nPos ;

    switch (LOWORD (wParam))
    {
    case SB_TOP:
        si.nPos = si.nMin ;
        break ;

    case SB_BOTTOM:
        si.nPos = si.nMax ;
        break ;

    case SB_LINEUP:
        si.nPos -= 1 ;
        break ;

    case SB_LINEDOWN:
        si.nPos += 1 ;
        break ;

    case SB_PAGEUP:
        si.nPos -= si.nPage ;
        break ;

    case SB_PAGEDOWN:
        si.nPos += si.nPage ;
        break ;

    case SB_THUMBTRACK:
        si.nPos = si.nTrackPos ;
        break ;

    default:
        break ;
    }
        // Set the position and then retrieve it. Due to adjustments
        // by Windows it might not be the same as the value set.
```

```
          si.fMask = SIF_POS ;
          SetScrollInfo (hwnd, SB_VERT, &si, TRUE) ;
          GetScrollInfo (hwnd, SB_VERT, &si) ;

               // If the position has changed, scroll the window and update it

          if (si.nPos != iVertPos)
          {
               ScrollWindow (hwnd, 0, cyChar * (iVertPos-si.nPos),
                                   NULL, NULL) ;
               UpdateWindow (hwnd) ;
          }
          return 0 ;

     case WM_HSCROLL:
               // Get all the vertical scroll bar information

          si.cbSize = sizeof (si) ;
          si.fMask  = SIF_ALL ;

               // Save the position for comparison later on

          GetScrollInfo (hwnd, SB_HORZ, &si) ;
          iHorzPos = si.nPos ;

          switch (LOWORD (wParam))
          {
          case SB_LINELEFT:
               si.nPos -= 1 ;
               break ;

          case SB_LINERIGHT:
               si.nPos += 1 ;
               break ;

          case SB_PAGELEFT:
               si.nPos -= si.nPage ;
               break ;

          case SB_PAGERIGHT:
               si.nPos += si.nPage ;
               break ;

          case SB_THUMBPOSITION:
               si.nPos = si.nTrackPos ;
               break ;
```

```
          default:
               break ;
          }
               // Set the position and then retrieve it. Due to adjustments
               // by Windows it might not be the same as the value set.

          si.fMask = SIF_POS ;
          SetScrollInfo (hwnd, SB_HORZ, &si, TRUE) ;
          GetScrollInfo (hwnd, SB_HORZ, &si) ;

               // If the position has changed, scroll the window

          if (si.nPos != iHorzPos)
          {
               ScrollWindow (hwnd, cxChar * (iHorzPos-si.nPos), 0,
                             NULL, NULL) ;
          }
          return 0 ;

     case WM_KEYDOWN:
          switch (wParam)
          {
          case VK_HOME:
               SendMessage (hwnd, WM_VSCROLL, SB_TOP, 0) ;
               break ;

          case VK_END:
               SendMessage (hwnd, WM_VSCROLL, SB_BOTTOM, 0) ;
               break ;

          case VK_PRIOR:
               SendMessage (hwnd, WM_VSCROLL, SB_PAGEUP, 0) ;
               break ;

          case VK_NEXT:
               SendMessage (hwnd, WM_VSCROLL, SB_PAGEDOWN, 0) ;
               break ;

          case VK_UP:
               SendMessage (hwnd, WM_VSCROLL, SB_LINEUP, 0) ;
               break ;
          case VK_DOWN:
               SendMessage (hwnd, WM_VSCROLL, SB_LINEDOWN, 0) ;
               break ;

          case VK_LEFT:
```

```
                SendMessage (hwnd, WM_HSCROLL, SB_PAGEUP, 0) ;
                break ;

            case VK_RIGHT:
                SendMessage (hwnd, WM_HSCROLL, SB_PAGEDOWN, 0) ;
                break ;
            }
            return 0 ;

        case WM_PAINT:
            hdc = BeginPaint (hwnd, &ps) ;

                // Get vertical scroll bar position

            si.cbSize = sizeof (si) ;
            si.fMask = SIF_POS ;
            GetScrollInfo (hwnd, SB_VERT, &si) ;
            iVertPos = si.nPos ;

                // Get horizontal scroll bar position

            GetScrollInfo (hwnd, SB_HORZ, &si) ;
            iHorzPos = si.nPos ;

                // Find painting limits

            iPaintBeg = max (0, iVertPos + ps.rcPaint.top / cyChar) ;
            iPaintEnd = min (NUMLINES - 1,
                             iVertPos + ps.rcPaint.bottom / cyChar) ;

            for (i = iPaintBeg ; i <= iPaintEnd ; i++)
            {
                x = cxChar * (1 - iHorzPos) ;
                y = cyChar * (i - iVertPos) ;

                TextOut (hdc, x, y,
                         sysmetrics[i].szLabel,
                         lstrlen (sysmetrics[i].szLabel)) ;

                TextOut (hdc, x + 22 * cxCaps, y,
                         sysmetrics[i].szDesc,
                         lstrlen (sysmetrics[i].szDesc)) ;

                SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;

                TextOut (hdc, x + 22 * cxCaps + 40 * cxChar, y, szBuffer,
                         wsprintf (szBuffer, TEXT ("%5d"),
```

```
                    GetSystemMetrics (sysmetrics[i].iIndex))) ;

               SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
          }

          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY:
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

6.3 字符消息

前面讨论了利用换档状态信息把击键消息翻译为字符消息的方法,并且提到,仅利用转换状态信息还不够,因为还需要知道同国家/地区有关的键盘配置。由于这个原因,您不应该试图把击键消息翻译为字符代码。Windows 会为您完成这一工作,在前面我们曾看到过以下代码:

```
while (GetMessage (&msg, NULL, 0, 0))
{
     TranslateMessage (&msg) ;
     DispatchMessage (&msg) ;
}
```

这是 WinMain 中典型的消息循环。GetMessage 函数用队列中的下一个消息填充 msg 结构的字段。DispatchMessage 以此消息为参数调用适当的窗口过程。

在这两个函数之间是 TranslateMessage 函数,它将击键消息转换为字符消息。如果消息为 WM_KEYDOWN 或者 WM_SYSKEYDOWN,并且击键与换档状态相组合产生一个字符,则 TranslateMessage 把字符消息放入消息队列中。此字符消息将是 GetMessage 从消息队列中得到的击键消息之后的下一个消息。

6.3.1 四类字符消息

字符消息可以分为四类,如表 6-9 所示。

WM_CHAR 和 WM_DEADCHAR 消息是从 WM_KEYDOWN 得到的;而 WM_SYSCHAR 和 WM_SYSDEADCHAR 消息是从 WM_SYSKEYDOWN 消息得到的。(我将简要地讨论一下什么是死字符。)

表 6-9

	字符	死字符
非系统字符:	WM_CHAR	WM_DEADCHAR
系统字符:	WM_SYSCHAR	WM_SYSDEADCHAR

有一个好消息:大多数情况下,Windows 程序会忽略除 WM_CHAR 之外的任何消息。伴随 4 个字符消息的 lParam 参数与产生字符代码消息的击键消息的 lParam 参数相同。不过,参数 wParam 不是虚拟键码。实际上,参数 wParam 是 ANSI 或 Unicode 字符代码。

这些字符消息是我们将文本传递给窗口过程时遇到的第一个消息。它们不是唯一的消息。其他消息伴随以 0 结尾的整个文本串。窗口过程是如何知道该字符是 8 位的 ANSI 还是 16 位的 Unicode 的呢?很简单:任何窗口过程都与您用 RegisterClassA(RegisterClass 的 ANSI 版)注册的窗口类相联系,从而获得含有 ANSI 字符代码的消息。如果窗口过程用 RegisterClassW(RegisterClass 的宽字符版)注册,那么传递给窗口过程的消息就带有 Unicode 字符代码。如果程序用 RegisterClass 注册窗口类,那么当定义 UNICODE 标识符时就调用 RegisterClassW,否则调用 RegisterClassA。

除非在编写程序的时候混合了 ANSI 和 Unicode 的函数与窗口过程,用 WM_CHAR 消息(及其他三种字符消息)说明的字符代码将是:

```
(TCHAR) wParam
```

同一个窗口过程可能会用到两个窗口类,一个用 RegisterClassA 注册,而另一个用 RegisterClassW 注册。也就是说,窗口过程可能会获得一些 ANSI 字符代码消息和一些 Unicode 字符代码消息。如果您的窗口过程需要排序输出,则它可以调用:

```
fUnicode = IsWindowUnicode (hwnd) ;
```

如果 hwnd 的窗口过程获得 Unicode 消息,那么变量 fUnicode 将为 TRUE,这表明窗口是基于用 RegisterClassW 注册的窗口类。

6.3.2 消息顺序

因为 TranslateMessage 函数从 WM_KEYDOWN 和 WM_SYSKEYDOWN 消息产生了字符消息,所以字符消息是夹在击键消息之间传递给窗口过程的。例如,如果 Caps Lock 未打开,而用户按下再释放 A 键,则窗口过程将接收到如表 6-10 所示的三个消息。

如果按下 Shift 键,再按下 A 键,然后释放 A 键,再释放 Shift 键,就会输入大写的 A,而窗口过程会接收到 5 个消息,如表 6-11 所示。Shift 键本身不产生字符消息。

表 6-10

消息	键或者代码
WM_KEYDOWN	"A"的虚拟键码(0x41)
WM_CHAR	"a"的字符代码(0x61)
WM_KEYUP	"A"的虚拟键码(0x41)

表 6-11

消息	键或者代码
WM_KEYDOWN	虚拟键码 VK_SHIFT (0x10)
WM_KEYDOWN	"A"的虚拟键码(0x41)
WM_CHAR	"A"的字符代码(0x41)
WM_KEYUP	"A"的虚拟键码(0x41)
WM_KEYUP	虚拟键码 VK_SHIFT(0x10)

如果用户按住 A 键,以使自动重复产生一系列的击键,那么对每个 WM_KEYDOWN 消息,都会得到一个字符消息,如表 6-12 所示。如果某些 WM_KEYDOWN 消息的重复计数大于 1,那么相应的 WM_CHAR 消息将具有同样的重复计数。使用 Ctrl 键与字母键的组合会产生 0x01(Ctrl-A)~0x1A(Ctrl-Z)的 ASCII 控制代码,其中的某些控制代码也可以由表 6-13 列出的键产生。最右列给出了在 ANSI C 中定义的控制符,它们用于描述这些键的字符代码。

表 6-12

消息	键或者代码
WM_KEYDOWN	"A"的虚拟键码(0x41)
WM_CHAR	"a"的字符代码(0x61)
WM_KEYDOWN	"A"的虚拟键码(0x41)
WM_CHAR	"a"的字符代码(0x61)
WM_KEYDOWN	"A"的虚拟键码(0x41)
WM_CHAR	"a"的字符代码(0x61)
WM_KEYDOWN	"A"的虚拟键码(0x41)
WM_CHAR	"a"的字符代码(0x61)
WM_KEYUP	"A"的虚拟键码(0x41)

有时 Windows 程序将 Ctrl 与字母键的组合用作菜单加速键(我将在第十章讨论),此时,不会将字母键转换成字符消息。

表 6-13

键	字符代码	产生方法	ANSI C 控制符
Backspace	0x08	Ctrl-H	\ b
Tab	0x09	Ctrl-I	\ t
Ctrl-Enter	0x0A	Ctrl-J	\ n
Enter	0x0D	Ctrl-M	\ r
Esc	0x1B	Ctrl-[	

6.3.3 处理控制字符

处理击键和字符消息的基本规则是：如果需要读取输入到窗口的键盘字符，那么您可以处理 WM_CHAR 消息。如果需要读取光标键、功能键、Delete 键、Insert 键、Shift 键、Ctrl 键，以及 Alt 键，那么您可以处理 WM_KEYDOWN 消息。

但是 Tab 键怎么办？Enter、Backspace 和 Escape 键又怎么办？传统上，这些键都产生表 6-13 列出的 ASCII 控制字符。但是在 Windows 中，它们也产生虚拟键码。这些键应该在处理 WM_CHAR 或者在处理 WM_KEYDOWN 期间处理吗？

经过 10 年的考虑（回顾这些年来我编写的 Windows 代码），我更喜欢将 Tab、Enter、Backspace 和 Escape 键处理成控制字符，而不是虚拟键。我通常这样处理 WM_CHAR：

```
case WM_CHAR:
     [other program lines]
     switch (wParam)
     {
     case '\b':                    // backspace
          [other program lines]
          break ;

     case '\t':                    // tab
          [other program lines]
          break ;

     case '\n':                    // linefeed
          [other program lines]
          break ;

     case '\r':                    // carriage return
          [other program lines]
          break ;

     default:                      // character codes
          [other program lines]
```

```
        break ;
    }
    return 0 ;
```

6.3.4 死字符消息

Windows 程序经常忽略 WM_DEADCHAR 和 WM_SYSDEADCHAR 消息，但您应该明确地知道死字符是什么，以及它们工作的方式。

在某些非 U.S.英语键盘上，有些键用于给字母加上音调。因为它们本身不产生字符，所以称之为“死键”。例如，当安装德国键盘时，对于 U.S.键盘上的 +/= 键，德国键盘的对应位置就是一个死键，当未按下 Shift 键时它用于标识锐音，当按下 Shift 键时它用于标识抑音。

当用户按下这个死键时，窗口过程接收到一个 wParam 等于音调本身的 ASCII 或者 Unicode 代码的 WM_DEADCHAR 消息。当用户再按下可以带有此音调的字母键(例如 A 键)时，窗口过程会接收到 WM_CHAR 消息，其中 wParam 等于带有音调的字母“a”的 ANSI 代码。

因此，用户程序不需要处理 WM_DEADCHAR 消息，原因是 WM_CHAR 消息已含有程序所需要的所有信息。Windows 逻辑甚至还构造了内部错误处理。如果在死键之后跟有不能带此音调符号的字母(例如“s”)，那么窗口过程将在一行接收到两个 WM_CHAR 消息——前一个消息的 wParam 等于音调符号本身的 ASCII 代码(同传递到 WM_DEADCHAR 消息的 wParam 值相同)，第二个消息的 wParam 等于字母 s 的 ASCII 代码。

当然，为找到感觉最好方法就是实际操作。您必须加载使用死键的外语键盘，例如前面讲过的德语键盘。可以这样设置：在“控制面板”中选择“键盘”，然后选择“语言”选项卡。然后您需要一个应用程序，该程序可以显示它接收的每一个键盘消息的详细信息。下面的 KEYVIEW1 就是这样的程序。

6.4 键盘消息和字符集

本章余下的样本程序有缺陷。它们不能在所有版本的 Windows 下都正常工作。这些缺陷不是特意引进代码中的；事实上，您也许永远不会遇到它们。这些问题——我不愿将它们称为“错误”——只有在不同的键盘语言和布局间切换时，以及在使用多字节字符集的远东版 Windows 下运行程序时，才显现出来。

不过，如果程序按 Unicode 编译并运行在 Windows NT 下，那么程序会运行得更好。我在第二章提到过这个问题，并且演示了 Unicode 对简化棘手的国际化问题的重要性。

6.4.1 KEYVIEW1 程序

了解键盘国际化问题的第一步就是检查 Windows 传递给窗口过程的键盘内容和字符消息。程序 6-2 所示的 KEYVIEW1 会对此有所帮助。该程序在客户区显示 Windows 向窗口过程发送的 8 种不同键盘消息的全部信息。

程序 6-2 KEYVIEW1 程序

KEYVIEW1.C

```
/*-----------------------------------------------------
   KEYVIEW1.C——Displays Keyboard and Character Messages
               (c) Charles Petzold, 1998
  -----------------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("KeyView1") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }
```

```
     hwnd = CreateWindow (szAppName, TEXT ("Keyboard Message Viewer # 1"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static int   cxClientMax, cyClientMax, cxClient, cyClient, cxChar, cyChar ;
     static int   cLinesMax, cLines ;
     static PMSG  pmsg ;
     static RECT  rectScroll ;
     static TCHAR szTop[] = TEXT ("Message        Key       Char     ")
                            TEXT ("Repeat Scan Ext ALT Prev Tran") ;
     static TCHAR szUnd[] = TEXT ("_____        __        ____       ")
                            TEXT ("____ __ __ __ ___ ___") ;
     static TCHAR * szFormat[2] = {

               TEXT ("%-13s %3d %-15s%c%6u %4d %3s %3s %4s %4s"),
               TEXT ("%-13s            0x%04X%1s%c %6u %4d %3s %3s %4s %4s") } ;
     static TCHAR * szYes  = TEXT ("Yes") ;
     static TCHAR * szNo   = TEXT ("No") ;
     static TCHAR * szDown = TEXT ("Down") ;
     static TCHAR * szUp   = TEXT ("Up") ;

     static TCHAR * szMessage [] = {
                         TEXT ("WM_KEYDOWN"),    TEXT ("WM_KEYUP"),
                         TEXT ("WM_CHAR"),       TEXT ("WM_DEADCHAR"),
                         TEXT ("WM_SYSKEYDOWN"), TEXT ("WM_SYSKEYUP"),
                         TEXT ("WM_SYSCHAR"), TEXT ("WM_SYSDEADCHAR") } ;
     HDC          hdc ;
     int          i, iType ;
     PAINTSTRUCT  ps ;
     TCHAR        szBuffer[128], szKeyName [32] ;
     TEXTMETRIC   tm ;

     switch (message)
```

```
     {
     case WM_CREATE:
     case WM_DISPLAYCHANGE:

               // Get maximum size of client area

          cxClientMax = GetSystemMetrics (SM_CXMAXIMIZED) ;
          cyClientMax = GetSystemMetrics (SM_CYMAXIMIZED) ;

               // Get character size for fixed-pitch font

          hdc = GetDC (hwnd) ;

          SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
          GetTextMetrics (hdc, &tm) ;
          cxChar = tm.tmAveCharWidth ;
          cyChar = tm.tmHeight ;

          ReleaseDC (hwnd, hdc) ;

               // Allocate memory for display lines

          if (pmsg)
               free (pmsg) ;

          cLinesMax = cyClientMax / cyChar ;
          pmsg = malloc (cLinesMax * sizeof (MSG)) ;
          cLines = 0 ;
                                        // fall through
     case WM_SIZE:
          if (message == WM_SIZE)
          {
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;
          }
               // Calculate scrolling rectangle

          rectScroll.left   = 0 ;
          rectScroll.right  = cxClient ;
          rectScroll.top    = cyChar ;
          rectScroll.bottom = cyChar * (cyClient / cyChar) ;

          InvalidateRect (hwnd, NULL, TRUE) ;
          return 0 ;

     case WM_KEYDOWN:
     case WM_KEYUP:
```

```
case WM_CHAR:
case WM_DEADCHAR:
case WM_SYSKEYDOWN:
case WM_SYSKEYUP:
case WM_SYSCHAR:
case WM_SYSDEADCHAR:

          // Rearrange storage array

     for (i = cLinesMax - 1 ; i > 0 ; i--)
     {
          pmsg[i] = pmsg[i - 1] ;
     }
          // Store new message

     pmsg[0].hwnd = hwnd ;
     pmsg[0].message = message ;
     pmsg[0].wParam = wParam ;
     pmsg[0].lParam = lParam ;

     cLines = min (cLines + 1, cLinesMax) ;

          // Scroll up the display

     ScrollWindow (hwnd, 0, -cyChar, &rectScroll, &rectScroll) ;

     break ;          // i.e., call DefWindowProc so Sys messages work
case WM_PAINT:
     hdc = BeginPaint (hwnd, &ps) ;

     SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
     SetBkMode (hdc, TRANSPARENT) ;
     TextOut (hdc, 0, 0, szTop, lstrlen (szTop)) ;
     TextOut (hdc, 0, 0, szUnd, lstrlen (szUnd)) ;

     for (i = 0 ; i < min (cLines, cyClient / cyChar - 1) ; i++)
     {
          iType = pmsg[i].message == WM_CHAR ||
                  pmsg[i].message == WM_SYSCHAR ||
                  pmsg[i].message == WM_DEADCHAR ||
                  pmsg[i].message == WM_SYSDEADCHAR ;

          GetKeyNameText (pmsg[i].lParam, szKeyName,
                          sizeof (szKeyName) / sizeof (TCHAR)) ;

          TextOut (hdc, 0, (cyClient / cyChar - 1 - i) * cyChar, szBuffer,
                  wsprintf (szBuffer, szFormat [iType],
```

```
                        szMessage [pmsg[i].message - WM_KEYFIRST],
                        pmsg[i].wParam,
                        (PTSTR) (iType ? TEXT (" ") : szKeyName),
                        (TCHAR) (iType ? pmsg[i].wParam : ' '),
                        LOWORD (pmsg[i].lParam),
                        HIWORD (pmsg[i].lParam) & 0xFF,
                        0x01000000 & pmsg[i].lParam ? szYes : szNo,
                        0x20000000 & pmsg[i].lParam ? szYes : szNo,
                        0x40000000 & pmsg[i].lParam ? szDown : szUp,
                        0x80000000 & pmsg[i].lParam ? szUp : szDown)) ;
          }
          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY:
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

KEYVIEW1 显示窗口过程接收到的每次击键和字符消息的内容，并将这些消息存储在一个 MSG 结构的数组中。该数组的大小基于最大化窗口的大小和等宽的系统字体。如果用户在程序运行时调整了视频显示的大小(在这种情况下 KEYVIEW1 接收 WM_DISPLAYCHANGE 消息)，将重新分配此数组。KEYVIEW1 使用标准 C 的 malloc 函数为数组分配内存。

下页图 6-2 给出了在键入单词“Windows”之后 KEYVIEW1 的屏幕显示。第一列显示了键盘消息；第二列在键名称的前面显示了击键消息的虚拟键代码，此代码通过 GetKeyNameText 函数获得；第三列(标注为“Char”)在字符本身的后面显示字符消息的十六进制字符代码。其余六列显示了 lParam 消息参数中 6 个字段的状态。

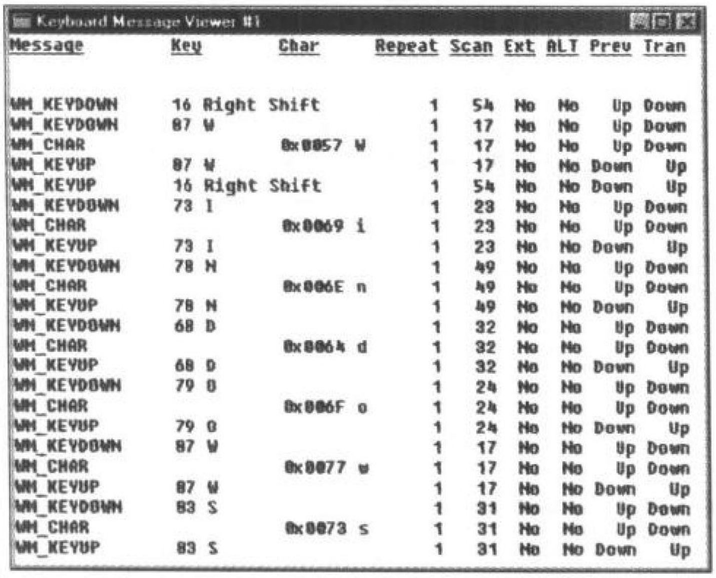
为便于以列的方式显示此信息，KEYVIEW1 使用了等宽字体。与前一章所讨论的一样，这需要调用 GetStockObject 和 SelectObject：

```
SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
```

KEYVIEW1 在客户区上部画了一个标题以确定 9 个列。此行文本带有下划线。虽然可以创建一种带下划线的字体，但这里使用了另一种方法。我定义了两个字符串变量 szTop(有文本)和 szUnd(有下划线)，并在 WM_PAINT 消息期间将它们同时显示在窗口顶部的同一位置。通常，Windows 以一种“不透明”的方式显示文本，也就是说显示字符时 Windows 将擦除字符背景区。这将导致第二个字符串(szUnd)擦除掉前一个(szTop)。要防止这一现象的发生，可将设备描述表切换到“透明”模式：

```
SetBkMode (hdc, TRANSPARENT) ;
```

这种加下划线的方法只有使用等宽字体时才可行。否则,下划线字符将无法与显现在下划线上面的字符等宽。



Keyboard Message Viewer #1

Message	Key	Char	Repeat	Scan	Ext	ALT	Prev	Tran
WM_KEYDOWN	16 Right Shift		1	54	No	No	Up	Down
WM_KEYDOWN	87 W		1	17	No	No	Up	Down
WM_CHAR		0x0057 W	1	17	No	No	Up	Down
WM_KEYUP	87 W		1	17	No	No	Down	Up
WM_KEYUP	16 Right Shift		1	54	No	No	Down	Up
WM_KEYDOWN	73 I		1	23	No	No	Up	Down
WM_CHAR		0x0069 i	1	23	No	No	Up	Down
WM_KEYUP	73 I		1	23	No	No	Down	Up
WM_KEYDOWN	78 N		1	49	No	No	Up	Down
WM_CHAR		0x006E n	1	49	No	No	Up	Down
WM_KEYUP	78 N		1	49	No	No	Down	Up
WM_KEYDOWN	68 D		1	32	No	No	Up	Down
WM_CHAR		0x0064 d	1	32	No	No	Up	Down
WM_KEYUP	68 D		1	32	No	No	Down	Up
WM_KEYDOWN	79 O		1	24	No	No	Up	Down
WM_CHAR		0x006F o	1	24	No	No	Up	Down
WM_KEYUP	79 O		1	24	No	No	Down	Up
WM_KEYDOWN	87 W		1	17	No	No	Up	Down
WM_CHAR		0x0077 w	1	17	No	No	Up	Down
WM_KEYUP	87 W		1	17	No	No	Down	Up
WM_KEYDOWN	83 S		1	31	No	No	Up	Down
WM_CHAR		0x0073 s	1	31	No	No	Up	Down
WM_KEYUP	83 S		1	31	No	No	Down	Up

图 6-2 KEYVIEW1 的屏幕显示

6.4.2 外语键盘问题

如果您运行美国英语版本的 Windows,那么您可以安装不同的键盘布局,并输入外语。可以在“控制面板”的“键盘”中安装外语键盘布局。选择“语言”选项卡,单击“添加”。要查看死键的工作方式,您可能想安装“德语”键盘。此外,我还要讨论“俄语”和“希腊语”的键盘布局,因此您也可安装这些键盘布局。如果在“键盘”显示的列表中找不到“俄语”和“希腊语”的键盘布局,则需要安装多语言支持:从“控制面板”中选择“添加/删除程序”,然后选择“Windows 安装程序”选项卡,确认选中“多语言支持”复选框。任何情况下,这些变更都需要原始 Windows 光盘。

安装完其他键盘布局后,您将在任务栏右侧的通知区看到一个带有两个字母代码的蓝色框。如果默认的是英语,那么这两个字母是“EN”。单击此图标,将得到所有已安装键盘布局的列表。从中单击需要的键盘布局可更改当前活动程序的键盘。此改变只影响当前活动的程序。

现在开始进行实验。不使用 UNICODE 标识符定义来编译 KEYVIEW1 程序。(在本书附带的光盘中,非 Unicode 版本的 KEYVIEW1 程序位于 RELEASE 子目录。)在美国英语版本的 Windows 下运行该程序,并输入字符“abcde”。WM_CHAR 消息与您所期望的一样:ASCII 字符代码 0x61、0x62、0x63、0x64 和 0x65,以及字母 a、b、c、d 和 e。

现在,KEYVIEW1 还在运行,选择德语键盘布局。按下 = 键,然后输入一个元音(a、

e、i、o 或者 u)。= 键将产生一条 WM_DEADCHAR 消息,元音产生一条 WM_CHAR 消息和(单独的)字符代码 0xE1、0xE9、0xED、0xF3、0xFA,以及字符 á、é、í、ó 和 ú。这就是死键的工作方式。

现在选择希腊键盘布局。输入"abcde",您会得到什么?您将得到 WM_CHAR 消息和字符代码 0xE1、0xE2、0xF8、0xE4、0xE5,以及字符 á、â、ø、ä 和 å。在这里有些字符不能正确显示。难道您不应该得到希腊字母表中的字母吗?

现在切换到俄语键盘并重新输入"abcde"。现在您得到 WM_CHAR 消息和字符代码 0xF4、0xE8、0xF1、0xE2 和 0xF3,以及字符 ô、è、ñ、â 和 ó。而且,还是有些字母不能正常显示。您应从斯拉夫字母表得到这些字母。

问题在于:您已经切换键盘以产生不同的字符代码,但您还没有将此切换通知 GDI,以便 GDI 能通过显示适当的符号来解释这些字符代码。

如果您非常勇敢,还有可供使用的共享 PC,并且有一个专业的或普通的 Microsoft 开发商网络(MSDN:Microsoft Developer Network)捐款,那么您也许希望安装(例如)希腊版的 Windows。您还可安装 4 种相同的键盘布局(英语、希腊语、德语和俄语)。现在运行 KEYLOOK1。切换到英语键盘布局,然后输入"abcde"。您应得到 ASCII 字符代码 0x61、0x62、0x63、0x64 和 0x65,以及字符 a、b、c、d 和 e。(并且您可放心:即使在希腊版,ASCII 还会正常运行。)

在希腊版的 Windows 中,切换到希腊键盘布局并输入"abcde"。您将得到 WM_CHAR 消息和字符代码 0xE1、0xE2、0xF8、0xE4 和 0xE5。这与您在安装希腊键盘布局的英语版 Windows 中得到的字符代码相同。但现在显示的字符是 α、β、Ψ、δ 和 ε。这些确实是小写的希腊字母 alpha、beta、psi、delta 和 epsilon。(gamma 怎么了?是这样,如果使用希腊版的 Windows,那么您将使用键帽上带有希腊字母的键盘。与英语 c 相对应的键正好是 psi。gamma 由与英语 g 相对应的键产生。您可在 Nadine Kano 编写的《Developing International Software for Windows 95 and Windows NT》的第 587 页看到完整的希腊字母表。

仍然在希腊版的 Windows 下运行 KEYVIEW1,切换到德语键盘布局。输入 = 键,然后依次输入 a、e、i、o 和 u。您将得到 WM_CHAR 消息和字符代码 0xE1、0xE9、0xED、0xF3 和 0xFA。这些字符代码与安装德语键盘布局的英语版 Windows 中的一样。不过,显示的字符却是 α、ι、ν、σ 和 ί,而不是正确的 á、é、í、ó 和 ú。

现在切换到俄语键盘并输入"abcde"。您会得到字符代码 0xF4、0xE8、0xF1、0xE2 和 0xF3,这与安装俄语键盘的英语版 Windows 中得到的一样。不过,显示的字符是 τ、θ、ρ、β 和 σ,而不是斯拉夫字母表中的字母。

您还可安装俄语版的 Windows。现在您可以猜到,英语和俄语键盘都可以工作,而德语和希腊语则不行。

现在,如果您真得很勇敢,您还可以安装日语版的 Windows,并运行 KEYVIEW1。如

果再按美国键盘输入，那么您将输入英语文本，一切都似乎正常。不过，如果切换到德语、希腊语或者俄语键盘布局，并且试着做上述介绍的任何练习，您将看到以点显示的字符。如果输入大写的字母——无论是带重音符号的德语字母、希腊语字母还是俄语字母——您将看到这些字母显示为日语中用于拼写外来语的片假名。您也许对输入片假名感兴趣，但那不是德语、希腊语或者俄语。

远东版本的 Windows 包括一个被称作"输入法编辑器"(IME)的实用程序，该程序显示为浮动的工具栏，它允许您用标准键盘输入象形文字，即汉语、日语和朝鲜语中使用的复杂字符。一般地，输入一组字母后，组成的字符将显示在另一个浮动窗口内。然后按 Enter 键，合成的字符代码就发送到活动窗口(即 KEYVIEW1)。KEYVIEW1 几乎没什么响应——WM_CHAR 消息拥有的字符代码多于 128 个，但这些代码没有意义。(Nadine Kano 书中有许多关于使用 IME 的内容。)

这时，我们已经看到了许多 KEYLOOK1 显示错误字符的例子——当运行安装了俄语或希腊语键盘布局的英语版 Windows 时，当运行安装了俄语或德语键盘布局的希腊版 Windows 时，以及当运行安装了德语、俄语或者希腊语键盘布局的俄语版 Windows 时，都是这样。我们也看到了从日语版 Windows 的输入法编辑器输入字符时的错误显示。

6.4.3 字符集和字体

KEYLOOK1 的问题是字体问题。用于在屏幕上显示字符的字体和键盘接收的字符代码不一致。因此，让我们看一下字体。

我将在第十七章进行详细讨论，Windows 支持三类字体——位图字体、矢量字体和(从 Windows 3.1 开始的)TrueType 字体。

事实上矢量字体已经过时了。这些字体中的字符由简单的线段组成，但这些线段没有定义填充区域。矢量字体可以较好地缩放到任意大小，但字符通常看上去有些单薄。

TrueType 字体是定义了填充区域的字符轮廓字体。TrueType 字体可缩放；而且该字符的定义包括"提示"，以回避可能带来的文本不可见或者不可读的圆整问题。使用 TrueType 字体，Windows 真正实现了 WYSIWYG("所见即所得")，即文本在视频显示器显示与打印机输出完全一致。

在位图字体中，每个字符都定义为与视频显示器上的像素对应的位点阵。位图字体可拉伸到较大的尺寸，但看上去带有锯齿。位图字体通常被设计者变形，以便在视频显示器上更容易阅读。因此，Windows 对标题栏、菜单、按钮和对话框中显示的文本都使用位图字体。

在默认的设备描述表下获得的位图字体称为系统字体。您可以通过调用带有

SYSTEM_FONT 标识符的 GetStockObject 函数来获得字体句柄。KEYVIEW1 程序选择使用 SYSTEM_FIXED_FONT 表示的等宽系统字体。GetStockObject 函数的另一个选项是 OEM_FIXED_FONT。

这三种字体有(各自的)字体名称——System、FixedSys 和 Terminal。程序可以在 CreateFont 或者 CreateFontIndirect 函数调用中使用字体名称来指定字体。这三种字体存储在 Windows 目录的 FONTS 子目录中的三个文件中的两个集合内。Windows 具体使用的文件集合取决于"控制面板"的"显示"中是选择显示"小字体"还是"大字体"(即您希望 Windows 假定视频显示器是 96 dpi 的分辨率还是 120 dpi 的分辨率)。表 6-14 总结了所有的情况。

表 6-14

GetStockObject 标识符	字体名称	小字体文件	大字体文件
SYSTEM_FONT	System	VGASYS.FON	8514SYS.FON
SYSTEM_FIXED_FONT	FixedSys	VGAFIX.FON	8514FIX.FON
OEM_FIXED_FONT	Terminal	VGAOEM.FON	8514OEM.FON

在文件名称中,"VGA"指的是视频图形阵列(Video Graphics Array),IBM 在 1987 年推出的视频适配器。这是 IBM 的第一块 PC 视频适配器,可显示 640×480 的像素大小。如果在"控制面板"的"显示"中选择了"小字体"(表示希望 Windows 假定视频显示的分辨率为 96 dpi),则 Windows 使用的这三种字体文件名将以"VGA"开头。如果选择了"大字体"(表示希望分辨率为 120 dpi),Windows 使用的文件名将以"8514"开头。8514 是 IBM 在 1987 年推出的另一种视频适配器,它的最大显示尺寸为 1024×768。

Windows 不希望您看到这些文件。这些文件的属性设置为系统和隐藏,如果用"Windows 资源管理器"来查看 FONTS 子目录的内容,您是不会看到它们的,即使选择了查看系统和隐藏文件也不行。从"工具"菜单选择"查找"选项来查找带有 *.FON 限制符的文件。这时,您可以双击文件名来查看字体字符是些什么。

对于许多标准控件和用户界面项目,Windows 不使用系统字体。相反,使用名称为 MS Sans Serif 的字体。("MS"代表 Microsoft。)这也是一种位图字体。文件(名为 SSERIFE.FON)包含基于 96 dpi 视频显示器的字体,磅值为 8、10、12、14、18 和 24。您可以在 GetStockObject 函数中使用 DEFAULT_GUI_FONT 标识符来得到该字体。Windows 使用的磅值取决于"控制面板"的"显示"中选择的显示分辨率。

到目前为止,我已提到 4 种标识符,利用这 4 种标识符,您可以用 GetStockObject 来获得用于设备描述表的字体。还有 3 种其他字体标识符:ANSI_FIXED_FONT、ANSI_VAR_FONT 和 DEVICE_DEFAULT_FONT。为了开始处理键盘和字符显示问题,让我们先看一下 Windows 中的所有备用字体。显示这些字体的程序是 STOKFONT,如程序6-3

所示。

程序 6-3 STOKFONT 程序

STOKFONT.C

```
/*----------------------------------------------
   STOKFONT.C -- Stock Font Objects
                 (c) Charles Petzold, 1998
  ----------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("StokFont") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Stock Fonts"),
                          WS_OVERLAPPEDWINDOW | WS_VSCROLL,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;
```

```
     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;
     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static struct
     {
          int     idStockFont ;
          TCHAR * szStockFont ;
     }
     stockfont [] = { OEM_FIXED_FONT,          "OEM_FIXED_FONT",
                      ANSI_FIXED_FONT,         "ANSI_FIXED_FONT",
                      ANSI_VAR_FONT,           "ANSI_VAR_FONT",
                      SYSTEM_FONT,             "SYSTEM_FONT",
                      DEVICE_DEFAULT_FONT,     "DEVICE_DEFAULT_FONT",
                      SYSTEM_FIXED_FONT,       "SYSTEM_FIXED_FONT",
                      DEFAULT_GUI_FONT,        "DEFAULT_GUI_FONT" } ;

     static int   iFont, cFonts = sizeof stockfont / sizeof stockfont[0] ;
     HDC          hdc ;
     int          i, x, y, cxGrid, cyGrid ;
     PAINTSTRUCT  ps ;
     TCHAR        szFaceName [LF_FACESIZE], szBuffer [LF_FACESIZE + 64] ;
     TEXTMETRIC   tm ;

     switch (message)
     {
     case WM_CREATE:
          SetScrollRange (hwnd, SB_VERT, 0, cFonts - 1, TRUE) ;
          return 0 ;

     case WM_DISPLAYCHANGE:
          InvalidateRect (hwnd, NULL, TRUE) ;
          return 0 ;

     case WM_VSCROLL:
          switch (LOWORD (wParam))
          {
          case SB_TOP:            iFont = 0 ;              break ;
          case SB_BOTTOM:         iFont = cFonts - 1 ;     break ;
```

```
        case SB_LINEUP:
        case SB_PAGEUP:          iFont -= 1 ;                break ;
        case SB_LINEDOWN:
        case SB_PAGEDOWN:        iFont += 1 ;                break ;
        case SB_THUMBPOSITION:   iFont = HIWORD (wParam) ;   break ;
        }
        iFont = max (0, min (cFonts - 1, iFont)) ;
        SetScrollPos (hwnd, SB_VERT, iFont, TRUE) ;
        InvalidateRect (hwnd, NULL, TRUE) ;
        return 0 ;

    case WM_KEYDOWN:
        switch (wParam)
        {
        case VK_HOME: SendMessage (hwnd, WM_VSCROLL, SB_TOP, 0) ;       break ;
        case VK_END:  SendMessage (hwnd, WM_VSCROLL, SB_BOTTOM, 0) ;    break ;
        case VK_PRIOR:
        case VK_LEFT:
        case VK_UP:   SendMessage (hwnd, WM_VSCROLL, SB_LINEUP, 0) ;    break ;
        case VK_NEXT:
        case VK_RIGHT:
        case VK_DOWN: SendMessage (hwnd, WM_VSCROLL, SB_PAGEDOWN, 0) ;  break ;
        }
        return 0 ;

    case WM_PAINT:
        hdc = BeginPaint (hwnd, &ps) ;

        SelectObject (hdc, GetStockObject (stockfont[iFont].idStockFont)) ;
        GetTextFace (hdc, LF_FACESIZE, szFaceName) ;
        GetTextMetrics (hdc, &tm) ;
        cxGrid = max (3 * tm.tmAveCharWidth, 2 * tm.tmMaxCharWidth) ;
        cyGrid = tm.tmHeight + 3 ;

        TextOut (hdc, 0, 0, szBuffer,
            wsprintf (szBuffer, TEXT (" %s: Face Name = %s, CharSet = %i"),
                      stockfont[iFont].szStockFont,
                      szFaceName, tm.tmCharSet)) ;

        SetTextAlign (hdc, TA_TOP | TA_CENTER) ;

                // vertical and horizontal lines

        for (i = 0 ; i < 17 ; i++)
        {
            MoveToEx (hdc, (i + 2) * cxGrid, 2 * cyGrid, NULL) ;
```

```
               LineTo   (hdc, (i + 2) * cxGrid, 19 * cyGrid) ;

               MoveToEx (hdc,      cxGrid, (i + 3) * cyGrid, NULL) ;
               LineTo   (hdc, 18 * cxGrid, (i + 3) * cyGrid) ;
          }
                // vertical and horizontal headings

          for (i = 0 ; i < 16 ; i++)
          {
               TextOut (hdc, (2 * i + 5) * cxGrid / 2, 2 * cyGrid + 2, szBuffer,
                    wsprintf (szBuffer, TEXT ("%X-"), i)) ;
               TextOut (hdc, 3 * cxGrid / 2, (i + 3) * cyGrid + 2, szBuffer,
                    wsprintf (szBuffer, TEXT ("-%X"), i)) ;
          }
                // characters

          for (y = 0 ; y < 16 ; y++)
          for (x = 0 ; x < 16 ; x++)
          {
               TextOut (hdc, (2 * x + 5) * cxGrid / 2,
                             (y + 3) * cyGrid + 2, szBuffer,
                    wsprintf (szBuffer, TEXT ("%c"), 16 * x + y)) ;
          }

          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY:
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

这个程序相当简单。它使用滚动条和光标移动键让您选择显示 7 种备用字体之一。该程序在一个网格中显示一种字体的 256 个字符。顶部的标题和网格的左侧显示字符代码的十六进制值。

在客户区域的顶部，STOKFONT 用 GetStockObject 函数显示用于选择字体的标识符。它还显示由 GetTextFace 函数得到的字体字样名称和 TEXTMETRIC 结构的 tmCharSet 字段。这个“字符集标识符”对理解 Windows 如何处理外语版本的 Windows 非常关键。

如果在美国英语版本的 Windows 中运行 STOKFONT，那么您看到的第一屏将显示使用 OEM_FIXED_FONT 标识符调用 GetStockObject 函数得到的字体。如图 6-3 所示。

在本字符集中（与本章其他部分一样），您将看到一些 ASCII。但请记住 ASCII 是 7 位代码，它定义了从代码 0x20 到 0x7E 的可显示字符。到 IBM 开发出 IBM PC 原型机

时,8位字节代码已被稳固地建立起来,因此可使用全8位代码作为字符代码。IBM决定使用一系列的有线和块组成的字符、带重音字母、希腊字母、数学符号和一些其他字符来扩展ASCII字符集。许多字符模式的MS-DOS程序在其屏幕显示中都使用线形字符,并且许多MS-DOS程序都在文件中使用了一些扩展字符。

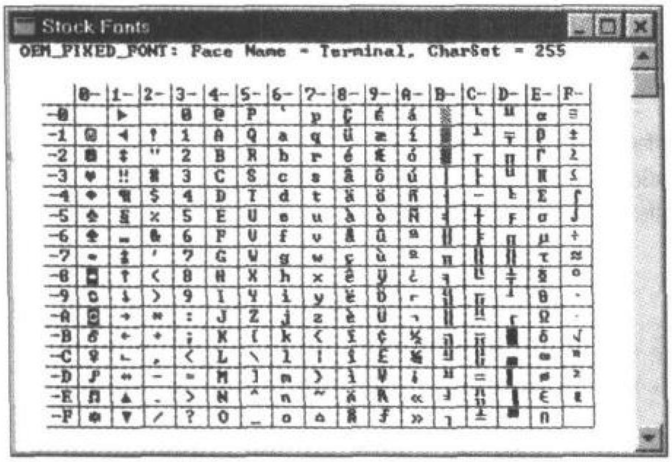


图6-3 美国版Windows中的OEM_FIXED_FONT

这个特殊的字符集给Windows的最初开发者带来了一个问题。一方面,因为Windows有完整的图形编程语言,所以线和块字符在Windows中不需要。因此,这些字符使用的48个代码最好用于许多西欧语言所需要的附带重音的字母。另一方面,IBM字符集定义了一个无法完全忽略的标准。

因此,Windows最初的开发者决定支持IBM字符集,但将其重要性降低到第二位——它们大多用于在窗口中运行的旧MS-DOS应用程序和需要使用由MS-DOS应用程序建立文件的Windows程序。Windows应用程序不使用IBM字符集,并且随着时间的推移,其重要性日渐衰退。然而,如果需要您还可以使用。在此环境下,"OEM"指的就是"IBM"。

(应知道外语版本的Windows不必支持与美国英语版相同的OEM字符集。其他国家有其自己的MS-DOS字符集。这完全是它自己的问题,而不是本书的问题。)

因为IBM字符集被认为不适合Windows,于是选择了另一种扩展字符集。此字符集被称作"ANSI字符集",但它实际上是ISO标准,也就是标准8859。它还被称为Latin 1、Western European,或者代码页1252。图6-4显示了ANSI字符集的一个版本——美国英语版Windows的系统字体。

粗的垂直条表示这些字符代码没有定义。注意,代码0x20到0x7E还是ASCII。此外,ASCII控制字符(0x00到0x1F以及0x7F)并不是可显示字符。它们本应如此。

代码0xC0到0xFF使得ANSI字符集对外语版Windows非常重要。这些代码提供64个在西欧语言中普遍使用的字符。字符0xA0,看起来像空格,但实际上定义为非断

开空格,例如"WW II"中的空格。

之所以说这是 ANSI 字符集的"一个版本",是因为存在代码 0x80 到 0x9F 的字符。等宽的系统字体只包括其中的两个字符,如图 6-5 所示。

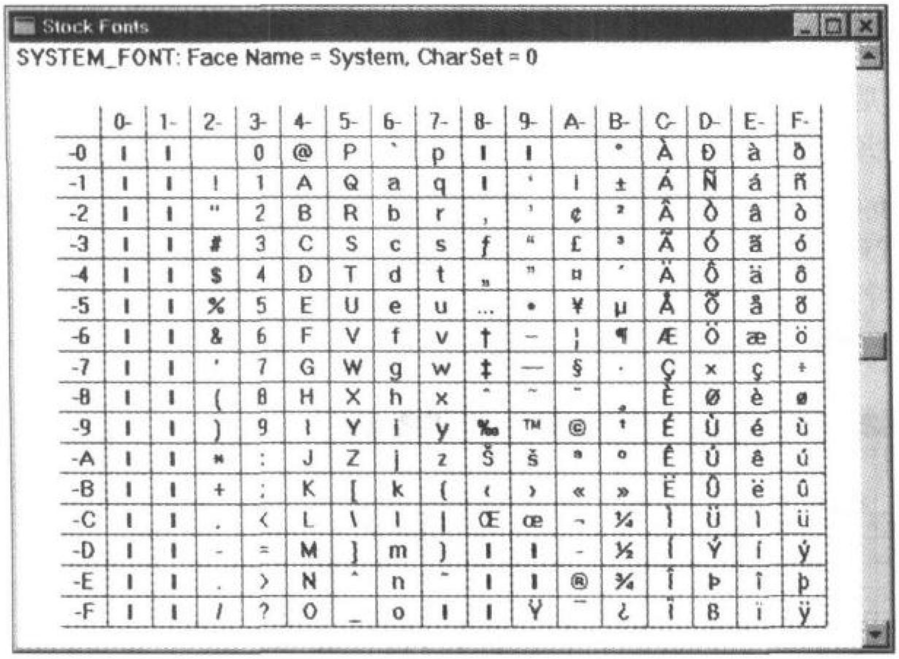


图 6-4　美国版 Windows 中的 SYSTEM_FONT

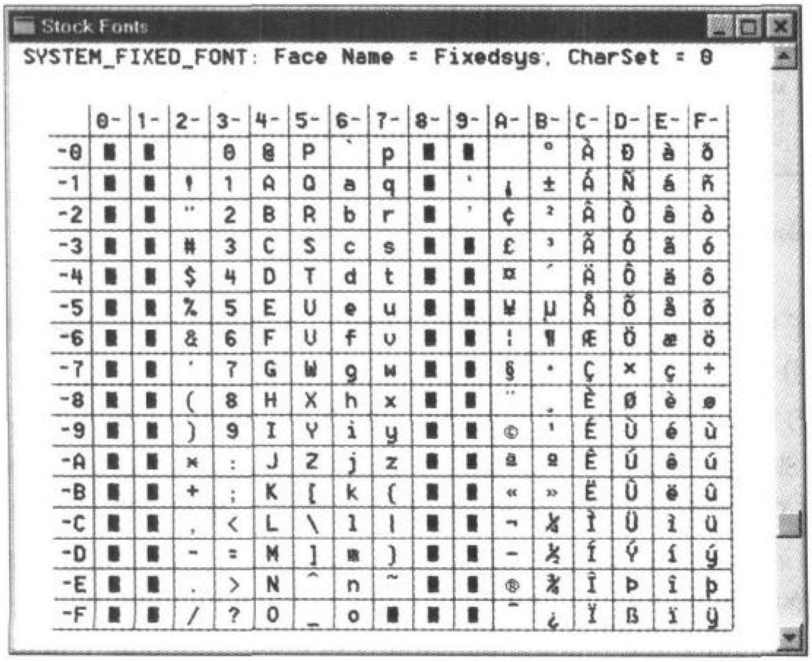


图 6-5　美国版 Windows 中的 SYSTEM_FIXED_FONT

在 Unicode 中,代码 0x0000 到 0x007F 与 ASCII 相同,代码 0x0080 到 0x009F 复制了 0x0000 到 0x001F 的控制字符,代码 0x00A0 到 0x00FF 与 Windows 中使用的 ANSI 字符集相同。

如果运行德语版的 Windows,那么当您用 SYSTEM_FONT 或者 SYSTEM_FIXED_FONT 标识符来调用 GetStockObject 函数时会得到同样的 ANSI 字符集。其他西欧版 Windows 也是如此。ANSI 字符集中含有这些语言所需要的所有字符。

不过,当您运行希腊版的 Windows 时,默认的字符集就改变了。相反,SYSTEM_FONT 如图6-6所示。SYSTEM_FIXED_FONT 有同样的字符。注意从 0xC0 到 0xFF 的代码。这些代码包含希腊字母表中的大写字母和小写字母。当您运行俄语版 Windows 时,默认的字符集如图 6-7 所示。此外,注意斯拉夫字母表中的大写和小写字母占用了代码 0xC0 和 0xFF。

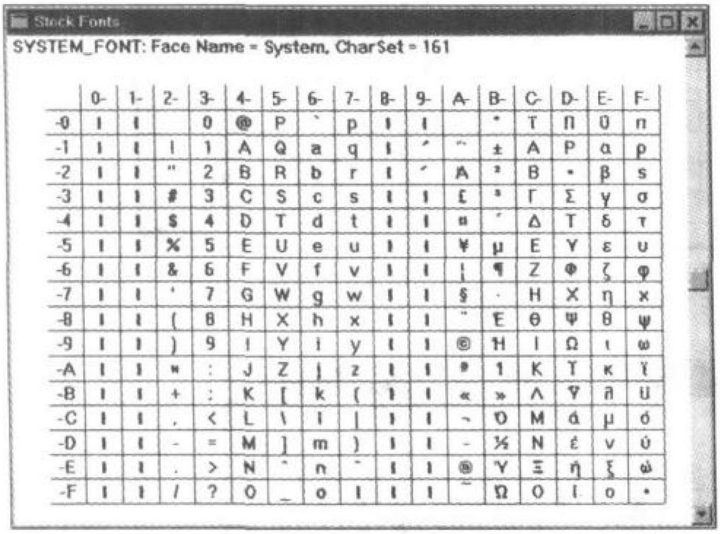


图 6-6　希腊版 Windows 中的 SYSTEM_FONT

图 6-8 显示了日语版 Windows 的 SYSTEM_FONT。从 0xA5 到 0xDF 的字符都是片假名字母表的一部分。

图 6-8 所示的日文系统字体不同于前面显示的那些,因为它实际上是双字节字符集(DBCS),称为"Shift-JIS"。("JIS"代表日本工业标准(Japanese Industrial Standard)。)从 0x81 到 0x9F 和从 0xE0 到 0xFF 的大多数字符代码实际上只是双字节代码的第一个字节,其第二个字节通常在 0x40 到 0xFC 的范围内。(关于这些代码的完整表格,请参见 Nadine Kano 书中的附录 G。)

现在,我们就可以看看 KEYVIEW1 中的问题在哪里:如果您安装了希腊键盘布局并键入"abcde",不考虑运行的 Windows 版本,Windows 将产生 WM_CHAR 消息和字符代

码 0xE1、0xE2、0xF8、0xE4 和 0xE5。但只有运行带有希腊系统字体的希腊版 Windows 时，这些字符代码才能与字符 α、β、Ψ、δ 和 ε 相对应。

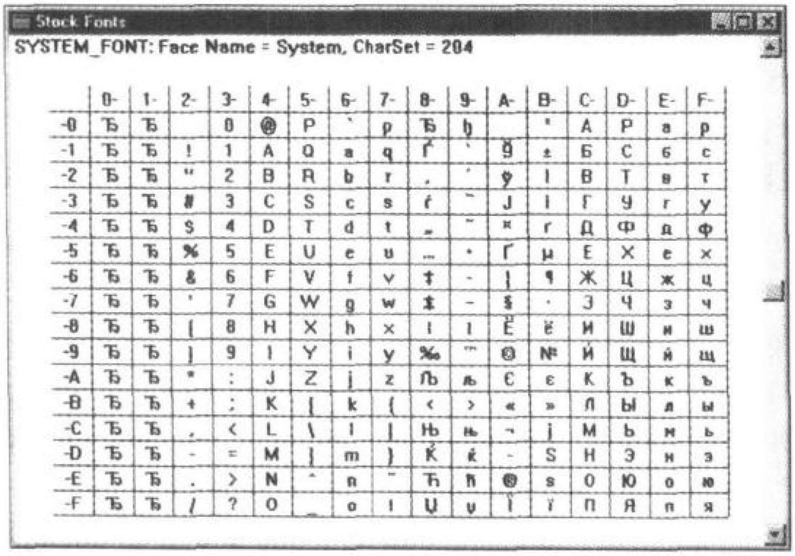


图 6-7 俄语版 Windows 中的 SYSTEM_FONT

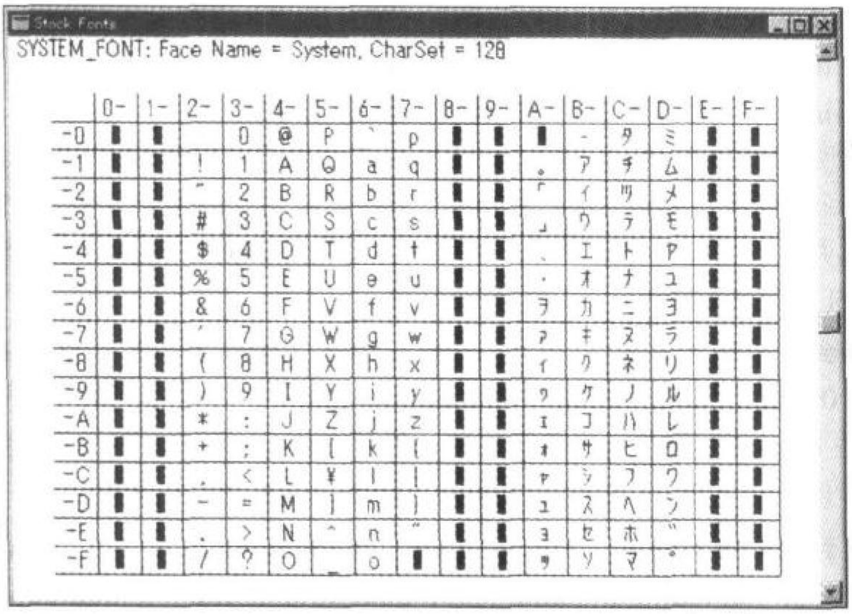


图 6-8 日语版 Windows 中的 SYSTEM_FONT

如果您安装了俄语键盘布局并键入“abcde”，不考虑所使用的 Windows 版本，Windows 将产生 WM_CHAR 消息和字符代码 0xF4、0xE8、0xF1、0xE2 和 0xF3。但只有在使用

俄语版 Windows 或者使用斯拉夫字母表的其他语言版，并且使用斯拉夫系统字体时，这些字符代码才会与字符 ф、и、с、в 和 у 相对应。

如果您安装了德语键盘布局并按下 = 键(或者位于同一位置的键)，然后按下 a、e、i、o 或者 u 键，不管运行何种 Windows 版本，Windows 都将产生 WM_CHAR 消息和字符代码 0xE1、0xE9、0xED、0xF3 和 0xFA。只有运行西欧版或者美国版的 Windows 时，也就是说有西欧系统字体，这些字符代码才会和字符 á、é、í、ó 或者 ú 相对应。

如果安装了美国英语键盘布局，则您可在键盘上键入任何字符，Windows 将产生 WM_CHAR 消息以及与字符正确匹配的字符代码。

6.4.4 Unicode 怎么样？

在第二章谈到过 Windows NT 支持的 Unicode 有助于为国际市场编写程序。让我们编译一下定义了 UNICODE 标识符的 KEYVIEW1，并在不同版本的 Windows NT 下运行。(在本书附带的光盘中，Unicode 版的 KEYVIEW1 位于 DEBUG 目录中。)

如果程序编译时定义了 UNICODE 标识符，则“KeyView1”窗口类就用 RegisterClassW 函数注册，而不是 RegisterClassA 函数。这意味着任何传递到 WndProc 的具有字符或文本数据的消息都将使用 16 位字符而不是 8 位字符。尤其是，WM_CHAR 消息将传递 16 位字符代码而不是 8 位字符代码。

在美国英语版的 Windows NT 下运行 Unicode 版的 KEYVIEW1。假定您已经安装了至少三种我们试验过的键盘布局——即德语、希腊语和俄语。

使用美国英语版的 Windows NT，并安装了英语或者德语的键盘布局，Unicode 版的 KEYVIEW1 在工作时将与非 Unicode 版相同。它将接收相同的字符代码(所有 0xFF 或者更低的值)，并显示同样正确的字符。这是因为最初的 256 个 Unicode 字符与 Windows 中使用的 ANSI 字符集相同。

现在切换到希腊键盘布局，并键入“abcde”。WM_CHAR 消息将含有 Unicode 字符代码 0x03B1、0x03B2、0x03C8、0x03B4 和 0x03B5。注意，我们先看到的字符代码值比 0xFF 高。这些 Unicode 字符代码与希腊字母 α、β、Ψ、δ 和 ε 相对应。不过，所有这 5 个字符都显示为方块！这是因为 SYSTEM_FIXED_FONT 只含有 256 个字符。

现在切换到俄语键盘布局，并键入“abcde”。KEYVIEW1 显示 WM_CHAR 消息和 Unicode 字符代码 0x0444、0x0438、0x0441、0x0432 和 0x0443，这些字符对应于斯拉夫字母 ф、и、с、в 和 у。不过，所有这 5 个字母也显示为实心方块。

简言之，非 Unicode 版的 KEYVIEW1 显示错误字符的地方，Unicode 版的 KEYVIEW1 就显示实心方块，以表示当前的字体没有那种特殊字符。虽然我不愿说 Unicode 版的 KEYVIEW1 是非 Unicode 版的改进，但事实确实如此。非 Unicode 版显示错误字符，而 Unicode 版不会这样。

Unicode 和非 Unicode 版 KEYVIEW1 的不同之处主要在两个方面。

首先，WM_CHAR 消息伴随一个 16 位字符代码，而不是 8 位字符代码。在非 Unicode 版本的 KEYVIEW1 中 8 位字符代码的含义取决于当前活动的键盘布局。如果来自德语键盘，则 0xE1 代码表示 ά，如果来自希腊语键盘则代表 α，如果来自俄语键盘则代表 ά。在 Unicode 版本程序中，16 位字符代码的含义很明确：σ 字符是 0x00E1，α 字符是 0x03B1，而 σ 字符是 0x0431。

第二，Unicode 的 TextOutW 函数显示的字符是基于 16 位字符的代码，而不是非 Unicode 的 TextOutA 函数的 8 位字符代码。因为这些 16 位字符代码含义明确，GDI 可以确定当前在设备描述表中选择的字体是否可显示每个字符。

在美国英语版 Windows NT 下运行 Unicode 版的 KEYVIEW1 多少让人感到有些迷惑，因为它所显示的就好像 GDI 只显示了 0x0000 到 0x00FF 之间的字符代码，而没有显示高于 0x00FF 的代码。也就是说，只是在字符代码和系统字体的 256 个字符之间简单的一对一的映射。

然而，如果安装了希腊或者俄语版的 Windows NT，您将发现情况就大不一样了。例如，如果安装了希腊版的 Windows NT，则美国英语、德语、希腊语和俄语键盘将会产生与美国英语版 Windows NT 同样的 Unicode 字符代码。不过，希腊版的 Windows NT 将不显示德语重音字符或者俄语字符，因为这些字符并不在希腊系统字体中。同样，俄语版的 Windows NT 也不显示德语重音字符或者希腊字符，因为这些字符也不在俄语系统字体中。

其中，Unicode 版的 KEYVIEW1 的区别在日语版 Windows NT 下更具戏剧性。您从 IME 输入日文字符，这些字符可以正确显示。唯一的问题是格式：因为日文字符通常看起来非常复杂，它们的显示宽度是其他字符的两倍。

6.4.5 TrueType 和大字体

我们使用的位图字体（在日文版 Windows 中带有附加字体）最多包括 256 个字符。这是我们所希望的，因为当假定字符代码是 8 位时，就返回到了早期的 Windows。这就是为什么当我们使用 SYSTEM_FONT 或者 SYSTEM_FIXED_FONT 时，某些语言中一些字符总不能正确显示。（日本系统字体有点不同，因为它是双字节字符集；大多数字符实际上保存在 TrueType 集合文件中，文件扩展名是.TCC。）

TrueType 字体包含的字符可以多于 256 个。并不是所有 TrueType 字体中的字符都多于 256 个，但 Windows 98 和 Windows NT 中的字体要包含多于 256 个字符。或者，安装了多语言支持后，TrueType 字体中也包含多于 256 个字符。在“控制面板”的“添加/删除程序”中，单击“Windows 安装程序”选项卡，并确保选中“多语言支持”。这个多语言支持包括 5 个字符集：波罗的语、中欧语、斯拉夫语、希腊语和土耳其语。波罗的语字符

集用于爱沙尼亚语、拉脱维亚语和立陶宛语。中欧字符集用于阿尔巴尼亚语、捷克语、克罗地亚语、匈牙利语、波兰语、罗马尼亚语、斯洛伐克语和斯洛文尼亚语。斯拉夫字符集用于保加利亚语、白俄罗斯语、俄语、塞尔维亚语和乌克兰语。

Windows 98 中的 TrueType 字体支持这 5 种字符集，再加上西欧(ANSI)字符集，西欧字符集实际上用于其他所有语言，但远东语言(汉语、日语和朝鲜语)除外。支持多种字符集的 TrueType 字体有时也称为“大字体”。在这种情况下的“大”并不是指字符的大小，而是指数量。

即使在非 Unicode 程序中也可利用大字体，这意味着可以用大字体显示几种不同字母表中的字符。然而，为把得到的字体选进设备描述表，还需要 GetStockObject 以外的函数。

函数 CreateFont 和 CreateFontIndirect 创建了一种逻辑字体，这与 CreatePen 建立逻辑笔以及 CreateBrush 建立逻辑画刷的方式类似。CreateFont 用 14 个参数描述要创建的字体。CreateFontIndirect 有一个参数，但该参数是指向 LOGFONT 结构的指针。LOGFONT 结构有 14 个字段，分别对应于 CreateFont 函数的参数。我将在第十七章详细讨论这些函数。现在，让我们看一下 CreateFont 函数，但我们只注意两个参数，其他参数都设置为 0。

如果需要等宽字体(就像 KEYVIEW1 程序中使用的)，将 CreateFont 的第 13 个参数设置为 FIXED_PITCH。如果需要非默认字符集的字体(这也是我们所需要的)，将 CreateFont 的第 9 个参数设置为某个“字符集 ID”。此字符集 ID 将是 WINGDI.H 中定义的下列值之一。我已给出注释，指出和这些字符集相关的代码页：

```
#define ANSI_CHARSET          0     // 1252 Latin 1 (ANSI)
#define DEFAULT_CHARSET       1
#define SYMBOL_CHARSET        2
#define MAC_CHARSET          77
#define SHIFTJIS_CHARSET    128     // 932 (DBCS, Japanese)
#define HANGEUL_CHARSET     129     // 949 (DBCS, Korean)
#define HANGUL_CHARSET      129     // "                 "
#define JOHAB_CHARSET       130     // 1361 (DBCS, Korean)
#define GB2312_CHARSET      134     // 936 (DBCS, Simplified Chinese)
#define CHINESEBIG5_CHARSET 136     // 950 (DBCS, Traditional Chinese)
#define GREEK_CHARSET       161     // 1253 Greek
#define TURKISH_CHARSET     162     // 1254 Latin 5 (Turkish)
#define VIETNAMESE_CHARSET  163     // 1258 Vietnamese
#define HEBREW_CHARSET      177     // 1255 Hebrew
#define ARABIC_CHARSET      178     // 1256 Arabic
#define BALTIC_CHARSET      186     // 1257 Baltic Rim
#define RUSSIAN_CHARSET     204     // 1251 Cyrillic (Slavic)
#define THAI_CHARSET        222     // 874 Thai
```

```
#define EASTEUROPE_CHARSET          238          // 1250 Latin 2 (Central Europe)
#define OEM_CHARSET                 255          // Depends on country
```

为什么 Windows 对同一个字符集有两个不同的数字:字符集 ID 和代码页 ID? 这只是 Windows 中的一种怪癖。注意,字符集 ID 只需要 1 字节的存储空间,这是 LOGFONT 结构中字符集字段的大小。(回忆 Windows 1.0 时期,内存和存储空间有限,每个字节都必须斤斤计较。)注意,有许多不同的 MS-DOS 代码页用于其他国家,但只有一种字符集 ID——OEM_CHARSET——用于 MS-DOS 字符集。

您还会注意到,这些字符集的值与 STOKFONT 程序顶行的"CharSet"值一致。在美国英语版 Windows 中,我们看到常备字体的字符集 ID 是 0(ANSI_CHARSET)和 255(OEM_CHARSET)。希腊版 Windows 中的是 161(GREEK_CHARSET),在俄语版中的是 204(RUSSIAN_CHARSET),在日语版中是 128(SHIFTJIS_CHARSET)。

在上面的代码中,DBCS 代表双字节字符集,用于远东版的 Windows。其他版的 Windows 不支持 DBCS 字体,因此不能使用那些字符集 ID。

CreateFont 返回 HFONT 值——逻辑字体的句柄。您可以使用 SelectObject 将此字体选进设备描述表。实际上,您必须调用 DeleteObject 来删除创建的所有逻辑字体。

大字体解决方案的其他部分是 WM_INPUTLANGCHANGE 消息。一旦您使用桌面弹出式菜单来改变键盘布局,Windows 都会向您的窗口过程发送 WM_INPUTLANGCHANGE 消息。wParam 消息参数是新键盘布局的字符集 ID。

程序 6-4 所示的 KEYVIEW2 程序实现了键盘布局改变时改变字体的逻辑。

程序 6-4 KEYVIEW2 程序

KEYVIEW2.C

```
/*--------------------------------------------------
   KEYVIEW2.C -- Displays Keyboard and Character Messages
                 (c) Charles Petzold, 1998
  --------------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("KeyView2") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;
```

```
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Keyboard Message Viewer #2"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static DWORD dwCharSet = DEFAULT_CHARSET ;
     static int   cxClientMax, cyClientMax, cxClient, cyClient, cxChar, cyChar ;
     static int   cLinesMax, cLines ;
     static PMSG  pmsg ;
     static RECT  rectScroll ;
     static TCHAR szTop[] = TEXT ("Message        Key       Char     ")
                            TEXT ("Repeat Scan Ext ALT Prev Tran") ;
     static TCHAR szUnd[] = TEXT ("_______        ___       ____     ")
                            TEXT ("______ ____ ___ ___ ____ ____") ;
```

```
static TCHAR * szFormat[2] = {

        TEXT ("%-13s %3d %-15s%c%6u %4d %3s %3s %4s %4s"),
        TEXT ("%-13s            0x%04X%1s%c %6u %4d %3s %3s %4s %4s") } ;

static TCHAR * szYes  = TEXT ("Yes") ;
static TCHAR * szNo   = TEXT ("No") ;
static TCHAR * szDown = TEXT ("Down") ;
static TCHAR * szUp   = TEXT ("Up") ;

static TCHAR * szMessage [] = {
                TEXT ("WM_KEYDOWN"),       TEXT ("WM_KEYUP"),
                TEXT ("WM_CHAR"),          TEXT ("WM_DEADCHAR"),
                TEXT ("WM_SYSKEYDOWN"),    TEXT ("WM_SYSKEYUP"),
                TEXT ("WM_SYSCHAR"),       TEXT ("WM_SYSDEADCHAR") } ;
HDC          hdc ;
int          i, iType ;
PAINTSTRUCT  ps ;
TCHAR        szBuffer[128], szKeyName [32] ;
TEXTMETRIC   tm ;

switch (message)
{
case WM_INPUTLANGCHANGE:
    dwCharSet = wParam ;
                                        // fall through
case WM_CREATE:
case WM_DISPLAYCHANGE:

        // Get maximum size of client area

cxClientMax = GetSystemMetrics (SM_CXMAXIMIZED) ;
cyClientMax = GetSystemMetrics (SM_CYMAXIMIZED) ;

        // Get character size for fixed-pitch font

hdc = GetDC (hwnd) ;

SelectObject (hdc, CreateFont (0, 0, 0, 0, 0, 0, 0, 0,
                  dwCharSet, 0, 0, 0, FIXED_PITCH, NULL)) ;

GetTextMetrics (hdc, &tm) ;
cxChar = tm.tmAveCharWidth ;
cyChar = tm.tmHeight ;

DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
ReleaseDC (hwnd, hdc) ;
```

```
          // Allocate memory for display lines

     if (pmsg)
          free (pmsg) ;

     cLinesMax = cyClientMax / cyChar ;
     pmsg = malloc (cLinesMax * sizeof (MSG)) ;
     cLines = 0 ;
                                        // fall through
case WM_SIZE:
     if (message == WM_SIZE)
     {
          cxClient = LOWORD (lParam) ;
          cyClient = HIWORD (lParam) ;
     }
          // Calculate scrolling rectangle

     rectScroll.left   = 0 ;
     rectScroll.right  = cxClient ;
     rectScroll.top    = cyChar ;
     rectScroll.bottom = cyChar * (cyClient / cyChar) ;

     InvalidateRect (hwnd, NULL, TRUE) ;

     if (message == WM_INPUTLANGCHANGE)
          return TRUE ;
     return 0 ;

case WM_KEYDOWN:
case WM_KEYUP:
case WM_CHAR:
case WM_DEADCHAR:
case WM_SYSKEYDOWN:
case WM_SYSKEYUP:
case WM_SYSCHAR:
case WM_SYSDEADCHAR:

          // Rearrange storage array

     for (i = cLinesMax - 1 ; i > 0 ; i--)
     {
          pmsg[i] = pmsg[i - 1] ;
     }
          // Store new message

     pmsg[0].hwnd = hwnd ;
```

```
          pmsg[0].message = message ;
          pmsg[0].wParam = wParam ;
          pmsg[0].lParam = lParam ;

          cLines = min (cLines + 1, cLinesMax) ;

               // Scroll up the display

          ScrollWindow (hwnd, 0, -cyChar, &rectScroll, &rectScroll) ;

          break ;        // ie, call DefWindowProc so Sys messages work

     case WM_PAINT:
          hdc = BeginPaint (hwnd, &ps) ;
          SelectObject (hdc, CreateFont (0, 0, 0, 0, 0, 0, 0, 0,
                              dwCharSet, 0, 0, 0, FIXED_PITCH, NULL)) ;

          SetBkMode (hdc, TRANSPARENT) ;
          TextOut (hdc, 0, 0, szTop, lstrlen (szTop)) ;
          TextOut (hdc, 0, 0, szUnd, lstrlen (szUnd)) ;

          for (i = 0 ; i < min (cLines, cyClient / cyChar - 1) ; i++)
          {
               iType = pmsg[i].message == WM_CHAR ||
                       pmsg[i].message == WM_SYSCHAR ||
                       pmsg[i].message == WM_DEADCHAR ||
                       pmsg[i].message == WM_SYSDEADCHAR ;

               GetKeyNameText (pmsg[i].lParam, szKeyName,
                              sizeof (szKeyName) / sizeof (TCHAR)) ;

               TextOut (hdc, 0, (cyClient / cyChar - 1 - i) * cyChar, szBuffer,
                        wsprintf (szBuffer, szFormat [iType],
                        szMessage [pmsg[i].message - WM_KEYFIRST],
                        pmsg[i].wParam,
                        (PTSTR) (iType ? TEXT (" ") : szKeyName),
                        (TCHAR) (iType ? pmsg[i].wParam : ' '),
                        LOWORD (pmsg[i].lParam),
                        HIWORD (pmsg[i].lParam) & 0xFF,
                        0x01000000 & pmsg[i].lParam ? szYes  : szNo,
                        0x20000000 & pmsg[i].lParam ? szYes  : szNo,
                        0x40000000 & pmsg[i].lParam ? szDown : szUp,
                        0x80000000 & pmsg[i].lParam ? szUp   : szDown)) ;
          }
          DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
          EndPaint (hwnd, &ps) ;
          return 0 ;
```

```
    case WM_DESTROY:
        PostQuitMessage (0) ;
        return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

注意,键盘输入语言改变后,KEYVIEW2 就清屏并重新分配存储空间。这样做有两条原因:第一,因为 KEYVIEW2 并不是某种字体专用的,当输入语言改变时字体字符的大小也会改变。程序需要根据新字符大小重新计算某些变量。第二,在接收每个字符消息时,KEYVIEW2 并不有效地保留字符集 ID。因此,如果键盘输入语言改变了,而且 KEYVIEW2 需要重画客户区时,所有的字符将用新字体显示。

第十七章将详细讨论字体和字符集。如果您想深入研究国际化问题,可以在/Platform SDK/Windows Base Services/International Features 找到需要的文档,但许多基础信息在/Platform SDK/Windows Base Services/General Library/String Manipulation 中都已提供。

6.5 插入符(不是光标)

当您往程序中输入文本时,通常有一个下划线、竖条或者方框来指示输入的下一个字符将出现在屏幕上的位置。该标志通常称为"光标",但在 Windows 下编程,您必须改变这个习惯。在 Windows 中,它称为"插入符"。"光标"是指表示鼠标位置的那个位图图像。

6.5.1 插入符函数

主要有 5 个插入符函数:

- CreateCaret 创建与窗口有关的插入符;
- SetCaretPos 在窗口中设置插入符的位置;
- ShowCaret 显示插入符;
- HideCaret 隐藏插入符;
- DestroyCaret 撤消插入符。

另外还有获取插入符当前位置(GetCaretPos)和获取与设置插入符闪烁时间(GetCaretBlinkTime 和 SetCaretBlinkTime)的函数。

在 Windows 中,插入符定义为水平线、与字符大小相同的方框,或者与字符同高的竖线。如果使用均衡字体,例如 Windows 默认的系统字体,则推荐使用竖线插入符。因为均衡字体中的字符没有固定大小,水平线或方框不能设置为字符的大小。

如果程序中需要插入符,那么您不应该简单地在窗口过程的 WM_CREATE 消息期间创建它,然后在 WM_DESTROY 消息期间撤消。其原因不用想也应该清楚:一个消息

队列只能支持一个插入符。因此，如果您的程序有多个窗口，那么各个窗口必须有效地共享相同的插入符。

其实，它并不像听起来那么受限制。当您考虑它的时候，只有当窗口有输入焦点时，窗口内显示插入符才有意义。事实上，闪烁的插入符只是一种视觉提示：您可以在程序中输入文本。因为任何时候都只有一个窗口拥有输入焦点，所以多个窗口同时都有闪烁的插入符是没有意义的。

通过处理 WM_SETFOCUS 和 WM_KILLFOCUS 消息，程序就可以确定它是否有输入焦点。正如名称所暗示的那样，窗口过程在有输入焦点的时候接收 WM_SETFOCUS 消息，失去输入焦点的时候接收 WM_KILLFOCUS 消息。这些消息成对出现：窗口过程在接收 WM_KILLFOCUS 消息之前将一直接收 WM_SETFOCUS 消息，并且在窗口打开期间，此窗口总是接收相同数量的 WM_SETFOCUS 和 WM_KILLFOCUS 消息。

使用插入符的主要规则很简单：窗口过程在 WM_SETFOCUS 消息期间调用 CreateCaret，在 WM_KILLFOCUS 消息期间调用 DestroyWindow。

这里还有几条其他的规则：插入符刚创建时是隐蔽的。如果想使插入符可见，那么您在调用 CreateCaret 之后，窗口过程还必须调用 ShowCaret。另外，当窗口过程处理一个非 WM_PAINT 消息而且希望在窗口内绘制某些东西时，它必须调用 HideCaret 隐藏插入符。在绘制完毕后，再调用 ShowCaret 显示插入符。HideCaret 的影响具有累积效果，如果多次调用 HideCaret 而不调用 ShowCaret，那么只有调用相同次数 ShowCaret 时，才能看到插入符。

6.5.2 TYPER 程序

程序 6-5 所示的 TYPER 程序使用了本章讨论的所有内容，您可以认为 TYPER 是一个特别简单的文本编辑器。在窗口中，您可以输入字符，用光标移动键(也可以称为插入符移动键)来移动光标(I 型标)，按下 Escape 键擦除窗口的内容等。缩放窗口、改变键盘输入语言时都会擦除窗口的内容。本程序没有滚动，没有查找和定位，不能保存文件，没有拼写检查，但它确实是一个开始。

程序 6-5 TYPER 程序

TYPER.C

```
/*-------------------
   TYPER.C——Typing Program
            (c) Charles Petzold, 1998
  -------------------*/
```

```
#include <windows.h>

#define BUFFER(x,y) *(pBuffer + y * cxBuffer + x)

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("Typer") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Typing Program"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}
```

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static DWORD   dwCharSet = DEFAULT_CHARSET ;
     static int     cxChar, cyChar, cxClient, cyClient, cxBuffer, cyBuffer,
                    xCaret, yCaret ;
     static TCHAR * pBuffer = NULL ;
     HDC            hdc ;
     int            x, y, i ;
     PAINTSTRUCT    ps ;
     TEXTMETRIC     tm ;

     switch (message)
     {
     case WM_INPUTLANGCHANGE:
     dwCharSet = wParam ;
                                   // fall through
     case WM_CREATE:
          hdc = GetDC (hwnd) ;
          SelectObject (hdc, CreateFont (0, 0, 0, 0, 0, 0, 0, 0,
                              dwCharSet, 0, 0, 0, FIXED_PITCH, NULL)) ;

          GetTextMetrics (hdc, &tm) ;
          cxChar = tm.tmAveCharWidth ;
          cyChar = tm.tmHeight ;

          DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
          ReleaseDC (hwnd, hdc) ;
                                   // fall through
     case WM_SIZE:
               // obtain window size in pixels

          if (message == WM_SIZE)
          {
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;
          }
               // calculate window size in characters

          cxBuffer = max (1, cxClient / cxChar) ;
          cyBuffer = max (1, cyClient / cyChar) ;

               // allocate memory for buffer and clear it

          if (pBuffer != NULL)
               free (pBuffer) ;

          pBuffer = (TCHAR *) malloc (cxBuffer * cyBuffer * sizeof (TCHAR)) ;
```

```
          for (y = 0 ; y < cyBuffer ; y++)
               for (x = 0 ; x < cxBuffer ; x++)
                    BUFFER(x,y) = ' ' ;

               // set caret to upper left corner

          xCaret = 0 ;
          yCaret = 0 ;

          if (hwnd == GetFocus ())
               SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;

          InvalidateRect (hwnd, NULL, TRUE) ;
          return 0 ;

     case WM_SETFOCUS:
                    // create and show the caret
          CreateCaret (hwnd, NULL, cxChar, cyChar) ;
          SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;
          ShowCaret (hwnd) ;
          return 0 ;

     case WM_KILLFOCUS:
                    // hide and destroy the caret

          HideCaret (hwnd) ;
          DestroyCaret () ;
          return 0 ;

     case WM_KEYDOWN:
          switch (wParam)
          {
          case VK_HOME:
               xCaret = 0 ;
               break ;

          case VK_END:
               xCaret = cxBuffer - 1 ;
               break ;

          case VK_PRIOR:
               yCaret = 0 ;
               break ;

          case VK_NEXT:
               yCaret = cyBuffer - 1 ;
               break ;
```

```
          case VK_LEFT:
               xCaret = max (xCaret - 1, 0) ;
               break ;

          case VK_RIGHT:
               xCaret = min (xCaret + 1, cxBuffer - 1) ;
               break ;

          case VK_UP:
               yCaret = max (yCaret - 1, 0) ;
               break ;

          case VK_DOWN:
               yCaret = min (yCaret + 1, cyBuffer - 1) ;
               break ;

          case VK_DELETE:
               for (x = xCaret ; x < cxBuffer - 1 ; x++ )
                    BUFFER (x, yCaret) = BUFFER (x + 1, yCaret) ;

               BUFFER (cxBuffer - 1, yCaret) = ' ' ;

               HideCaret (hwnd) ;
               hdc = GetDC (hwnd) ;

               SelectObject (hdc, CreateFont (0, 0, 0, 0, 0, 0, 0, 0,
                                   dwCharSet, 0, 0, 0, FIXED_PITCH, NULL)) ;

               TextOut (hdc, xCaret * cxChar, yCaret * cyChar,
                        & BUFFER (xCaret, yCaret),
                        cxBuffer - xCaret) ;
               DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
               ReleaseDC (hwnd, hdc) ;
               ShowCaret (hwnd) ;
               break ;
          }
          SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;
          return 0 ;

     case WM_CHAR:
          for (i = 0 ; i < (int) LOWORD (lParam) ; i++ )
          {
               switch (wParam)
               {
               case '\b':                    // backspace
                    if (xCaret > 0)
                    {
```

```
                    xCaret-- ;
                    SendMessage (hwnd, WM_KEYDOWN, VK_DELETE, 1) ;
               }
               break ;

          case '\t':                         // tab
               do
               {
                    SendMessage (hwnd, WM_CHAR, ' ', 1) ;
               }
               while (xCaret % 8 != 0) ;
               break ;
          case '\n':                         // line feed
               if (++yCaret == cyBuffer)
                    yCaret = 0 ;
               break ;

          case '\r':                         // carriage return
               xCaret = 0 ;

               if (++yCaret == cyBuffer)
                    yCaret = 0 ;
               break ;

          case '\x1B':                         // escape
               for (y = 0 ; y < cyBuffer ; y++)
                    for (x = 0 ; x < cxBuffer ; x++)
                         BUFFER (x, y) = ' ' ;

               xCaret = 0 ;
               yCaret = 0 ;

               InvalidateRect (hwnd, NULL, FALSE) ;
               break ;

          default:                                   // character codes
               BUFFER (xCaret, yCaret) = (TCHAR) wParam ;

               HideCaret (hwnd) ;
               hdc = GetDC (hwnd) ;

               SelectObject (hdc, CreateFont (0, 0, 0, 0, 0, 0, 0, 0,
                              dwCharSet, 0, 0, 0, FIXED_PITCH, NULL)) ;
               TextOut (hdc, xCaret * cxChar, yCaret * cyChar,
                        & BUFFER (xCaret, yCaret), 1) ;
               DeleteObject (
                    SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
               ReleaseDC (hwnd, hdc) ;
```

```
                    ShowCaret (hwnd) ;

                    if ( ++ xCaret == cxBuffer)
                    {
                         xCaret = 0 ;

                         if ( ++ yCaret == cyBuffer)
                              yCaret = 0 ;
                    }
                    break ;
               }
          }

          SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;
          return 0 ;

     case WM_PAINT:
          hdc = BeginPaint (hwnd, &ps) ;

          SelectObject (hdc, CreateFont (0, 0, 0, 0, 0, 0, 0, 0,
                                   dwCharSet, 0, 0, 0, FIXED_PITCH, NULL)) ;

          for (y = 0 ; y < cyBuffer ; y++ )
               TextOut (hdc, 0, y * cyChar, & BUFFER(0,y), cxBuffer) ;
          DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY:
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

为了简单起见，TYPER 程序使用一种等宽字体，因为编写处理均衡字体的文本编辑器要困难得多。程序可以在好几个地方获取设备描述表：在 WM_CREATE 消息期间，在 WM_KEYDOWN 消息期间，在 WM_CHAR 消息期间以及在 WM_PAINT 消息期间，每次都可以通过 GetStockObject 和 SelectObject 调用来选择等宽字体。

在 WM_SIZE 消息期间，TYPER 计算窗口的字符宽度和高度并把值保存在 cxBuffer 和 cyBuffer 变量中，然后使用 malloc 分配缓冲区以保存在窗口内输入的所有字符。注意，缓冲区的字节大小取决于 cxBuffer、cyBuffer 和 sizeof(TCHAR)，它可以是 1 或 2，这依赖于程序是以 8 位的字符处理还是以 Unicode 方式编译的。

xCaret 和 yCaret 变量保存插入符位置。在 WM_SETFOCUS 消息期间，TYPER 调用

CreateCaret 来创建与字符有相同宽度和高度的插入符，调用 SetCaretPos 来设置插入符的位置，调用 ShowCaret 使插入符可见。在 WM_KILLFOCUS 消息期间，TYPER 调用 HideCaret 和 DestroyCaret。

对 WM_KEYDOWN 的处理大多要涉及光标移动键，另外，Home 和 End 把插入符送至一行的开始和末尾处，Page Up 和 Page Down 把插入符送至窗口的顶端和底部，箭头的用法不变。对 Delete 键，TYPER 将缓冲区中从插入符之后的那个位置开始到行尾的所有内容向前移动，并在行尾显示空格。

WM_CHAR 处理 Backspace、Tab、Linefeed(Ctrl-Enter)、Enter、Escape 和字符键。注意，在处理 WM_CHAR 消息时(假设用户输入的每个字符都非常重要)，我使用了 lParam 中的重复计数；而在处理 WM_KEYDOWN 消息时却不这么做(禁止有害的重复滚动)。对 Backspace 和 Tab 的处理由于使用了 SendMessage 函数而得到简化，Backspace 由 Delete 来模拟，而 Tab 由若干个空格来模拟。

前面我已经提到过，在非 WM_PAINT 消息期间，如果要在窗口中绘制内容，则应该隐蔽光标。TYPER 为 Delete 键处理 WM_KEYDOWN 消息和为字符键处理 WM_CHAR 消息时即是如此。在这两种情况下，TYPER 改变缓冲区中的内容，然后在窗口中绘制一个或者多个新字符。

虽然 TYPER 使用了与 KEYVIEW2 相同的逻辑以在字符集之间切换(就像用户切换键盘布局一样)，但对于远东版的 Windows，它还是不能正常工作。TYPER 不允许两倍宽度的字符。此问题将在第十七章讨论，那时我们将详细讨论字体与文本输出。

第七章

鼠标

鼠标是有一个或多个键的定位设备。虽然也可以使用诸如触摸屏和光笔之类的输入设备,但是只有鼠标以及常用在膝上计算机上的跟踪球等才是渗透了 PC 市场的唯一输入设备。

情况并非总是如此。当然,Windows 的早期开发人员认为他们不应该要求用户为了运行其产品而必须买只鼠标。因此,他们将鼠标作为一种可选的附加设备,而为 Windows 中的所有操作以及"小应用程序"提供一种键盘接口。(例如,查看 Windows 的计算器的帮助信息,可以看到每个按钮都提供了一个等价的键盘操作。)第三方软件开发人员在他们的应用程序中使用键盘接口来模拟鼠标的功能,并且这本书以前的版本也是这么做的。

从理论上讲,现在的 Windows 需要鼠标。至少,一些消息框是这样讲的。当然,您也可以拔下鼠标,而且 Windows 也会运行良好(只有消息框会提示您没有连接鼠标)。试图不用鼠标来使用 Windows 就像用脚趾来弹钢琴一样(至少在最初的一段时间里是这样),但您依然可以这样做。正因为如此,我还是喜欢为鼠标功能提供键盘操作。打字员尤其喜欢让他们的手保持在键盘上,并且我认为每个人都有在杂乱的桌上找不到鼠标或者鼠标移动不灵敏的经历。使用键盘通常不需要花费更多的精力和努力,并且为喜欢使用键盘的人提供更多的功能。

我们通常认为,键盘便于输入和操作文本数据,而鼠标则便于画图和操作图形对象。实际上,本章大多数的示例程序都画了一些图形,并且用到了我们在第五章所学到的知识。

7.1 鼠标基础

Windows 98 能支持单键、双键或者三键鼠标,也可以使用操纵杆或者光笔来模拟单键鼠

标。早期,由于许多用户都有单键鼠标,所以 Windows 应用程序总是避免使用双键或三键鼠标。不过,由于双键鼠标已经成为事实上的标准,因而不使用第二个键的传统已经不再合理了。当然,第二个鼠标键是用于激活一个"上下文菜单",即出现在普通菜单之外的窗口中的菜单,或者用于特殊的拖动操作(拖动将在后面加以解释)。然而,程序不能依赖双键鼠标。

理论上讲,您可以用我们的老朋友 GetSystemMetrics 函数来确定鼠标是否存在:

```
fMouse = GetSystemMetrics (SM_MOUSEPRESENT) ;
```

如果已经安装了鼠标,fMouse 将返回 TRUE(非 0);如果没有安装,则返回 0。然而,在 Windows 98 中,不论鼠标是否安装,此函数都将返回 TRUE 。在 Microsoft Windows NT 中,它可以正常工作。

要确定所安装鼠标上键的个数,可使用

```
cButtons = GetSystemMetrics (SM_CMOUSEBUTTONS) ;
```

如果没有安装鼠标,那么函数将返回 0。然而,在 Windows 98 下,如果没有安装鼠标,此函数将返回 2。

习惯用左手的用户可以使用 Windows 的"控制面板"来切换鼠标键。虽然应用程序可以通过在 GetSystemMetrics 中使用 SM_SWAPBUTTON 参数来确定是否进行了这种切换,但通常没有这个必要。由食指触发的键被认为是左键,即使事实上是位于鼠标的右边。不过,在一个培训程序中,您可能想在屏幕上画一个鼠标,在这种情况下,您可能想知道鼠标键是否被切换过了。

您能够在"控制面板"中设置鼠标的其他参数,比如双击速度。从 Windows 应用程序,您能够通过使用 SystemParametersInfo 函数设置或获得此信息。

7.1.1 一些简单的定义

当 Windows 用户移动鼠标时,Windows 在显示屏上移动一个称为"鼠标光标"的小位图。鼠标光标有一个指向显示屏上精确位置的单像素的"热点"。当我提到鼠标光标在屏幕上的位置时,指的是热点的位置。

Windows 支持几种预定义的鼠标光标,程序可以使用这些光标。最常见的是称为 IDC_ARROW 的斜箭头(在 WINUSER.H 中定义)。热点在箭头的顶端。IDC_CROSS 光标(在本章后面的 BLOKOUT 程序中用到)的热点在十字交叉线的中心。IDC_WAIT 光标是一个时钟,通常用于指示程序正在运行。程序员也可以设计自己的光标。我们将在第十章学习设计方法。当定义窗口类结构时指定特定窗口的默认光标,例如:

```
wndclass.hCursor = LoadCursor (NULL, IDC_ARROW) ;
```

下面是一些描述鼠标键动作的术语：

- **单击**　按下并放开一个鼠标键；
- **双击**　快速按下并放开鼠标键两次；
- **拖拽**　按住鼠标键并移动鼠标。

对三键鼠标，三个键分别被称为左键、中键、右键。在 Windows 头文件中定义的与鼠标有关的标识符使用缩写 LBUTTON、MBUTTON 和 RBUTTON。双键鼠标只有左键与右键，单键鼠标只有一个左键。

7.2　客户区鼠标消息

在前一章中您已经看到，Windows 只把键盘消息发送给拥有输入焦点的窗口。鼠标消息与此不同：只要鼠标跨越窗口或者在某窗口中按下鼠标键，那么窗口过程就会收到鼠标消息，而不管该窗口是否活动或者是否拥有输入焦点。Windows 为鼠标定义了 21 种消息，不过，其中有 11 种消息同客户区无关（下面称之为"非客户区"消息），Windows 程序经常忽略这些消息。

当鼠标移过窗口的客户区时，窗口过程收到 WM_MOUSEMOVE 消息。当在窗口的客户区中按下或者释放一个鼠标键时，窗口过程会接收到如表 7-1 所示的消息。

表 7-1

键	按下	释放	双击键
左	WM_LBUTTONDOWN	WM_LBUTTONUP	WM_LBUTTONDBLCLK
中	WM_MBUTTONDOWN	WM_MBUTTONUP	WM_MBUTTONDBLCLK
右	WM_RBUTTONDOWN	WM_RBUTTONUP	WM_RBUTTONDBLCLK

只有对三键鼠标，窗口过程才会收到 MBUTTON 消息；只有对双键或者三键鼠标，才会接收到 RBUTTON 消息。仅当定义的窗口类能接收 DBLCLK（双击）消息之后，窗口过程才能接收到这些消息（请参见 7.2.3"双击鼠标键"一节）。

对于所有这些消息来说，其 lParam 值均含有鼠标的位置：低位字为 x 坐标，高位字为 y 坐标，这两个坐标是相对于窗口客户区左上角的位置。您可以用 LOWORD 和 HIWORD 宏来提取这些值：

```
x = LOWORD (lParam) ;
y = HIWORD (lParam) ;
```

wParam 的值指示鼠标键及 Shift 和 Ctrl 键的状态。您可以使用头文件 WINUSER.H 中定义的位屏蔽来测试 wParam。MK 前缀代表"鼠标键"。

MK_LBUTTON	按下左键
MK_MBUTTON	按下中键
MK_RBUTTON	按下右键
MK_SHIFT	按下 Shift 键
MK_CONTROL	按下 Ctrl 键

例如,如果收到了 WM_LBUTTONDOWN 消息,而且值

```
wparam & MK_SHIFT
```

是 TRUE(非 0),您就知道当左键按下时也按下了 Shift 键。

当您把鼠标移过窗口的客户区时,Windows 并不为鼠标的每个可能的像素位置都产生一条 WM_MOUSEMOVE 消息。您的程序接收到 WM_MOUSEMOVE 消息的次数依赖于鼠标硬件以及您的窗口过程处理鼠标移动消息的速度。换句话说,Windows 不能用未处理的 WM_MOUSEMOVE 消息来填充消息队列。当您运行下面将描述的 CONNECT 程序时,您将会更好地理解 WM_MOUSEMOVE 消息的速率。

如果您在非活动窗口的客户区中按下鼠标左键,Windows 将把活动窗口改为在其中按下鼠标键的窗口,然后把 WM_LBUTTONDOWN 消息送到该窗口过程。当窗口过程得到 WM_LBUTTONDOWN 消息时,您的程序就可以安全地假定该窗口是活动的了。不过,您的窗口过程可能在未接收到 WM_LBUTTONDOWN 消息的情况下先接收到了 WM_LBUTTONUP 的消息。如果在一个窗口中按下鼠标键,然后移动到用户窗口释放它,就会出现这种情况。类似地,当鼠标键在另一个窗口中被释放时,窗口过程只能接收到 WM_LBUTTONDOWN 消息而没有相应的 WM_LBUTTONUP 消息。

这些规则有两个例外:

- 窗口过程可以"捕获鼠标"并且连续地接收鼠标消息,即使此时鼠标在该窗口客户区之外。您将在本章的后面学习如何捕获鼠标。
- 如果正在显示一个模态系统消息框或者模态系统对话框,那么其他程序就不能接收鼠标消息。当模态系统消息框或者对话框处于活动态时,禁止切换到其他窗口或者程序。当您关闭 Windows 会话时,将会显示一个模态系统消息框的例子。

7.2.1 简单的鼠标处理:一个例子

程序 7-1 中所示的 CONNECT 程序能做一些简单的鼠标处理,使您对 Windows 如何向程序发送鼠标消息有一些体会。

程序 7-1 CONNECT 程序

CONNECT.C

```
/*-------------------------------------------------
   CONNECT.C——Connect-the-Dots Mouse Demo Program
              (c) Charles Petzold, 1998
  -------------------------------------------------*/

#include <windows.h>

#define MAXPOINTS 1000
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("Connect") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Connect-the-Points Mouse Demo"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;
```

```
          ShowWindow (hwnd, iCmdShow) ;
          UpdateWindow (hwnd) ;

          while (GetMessage (&msg, NULL, 0, 0))
          {
               TranslateMessage (&msg) ;
               DispatchMessage (&msg) ;
          }

     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
     {
     static POINT      pt[MAXPOINTS] ;
     static int        iCount ;
     HDC               hdc ;
     int               i, j ;
     PAINTSTRUCT       ps ;

     switch (message)
          {
     case WM_LBUTTONDOWN:
          iCount = 0 ;
          InvalidateRect (hwnd, NULL, TRUE) ;
          return 0 ;

     case WM_MOUSEMOVE:
          if (wParam & MK_LBUTTON && iCount < 1000)
          {
               pt[iCount   ].x = LOWORD (lParam) ;
               pt[iCount ++ ].y = HIWORD (lParam) ;

               hdc = GetDC (hwnd) ;
               SetPixel (hdc, LOWORD (lParam), HIWORD (lParam), 0) ;
               ReleaseDC (hwnd, hdc) ;
          }
          return 0 ;

     case WM_LBUTTONUP:
          InvalidateRect (hwnd, NULL, FALSE) ;
          return 0 ;

     case WM_PAINT:
          hdc = BeginPaint (hwnd, &ps) ;

          SetCursor (LoadCursor (NULL, IDC_WAIT)) ;
          ShowCursor (TRUE) ;
```

```
        for (i = 0 ; i < iCount - 1 ; i++ )
            for (j = i + 1 ; j < iCount ; j++ )
            {
                MoveToEx (hdc, pt[i].x, pt[i].y, NULL) ;
                LineTo   (hdc, pt[j].x, pt[j].y) ;
            }

        ShowCursor (FALSE) ;
        SetCursor (LoadCursor (NULL, IDC_ARROW)) ;
        EndPaint (hwnd, &ps) ;
        return 0 ;

    case WM_DESTROY:
        PostQuitMessage (0) ;
        return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

CONNECT 处理三个鼠标消息：

- **WM_LBUTTONDOWN** CONNECT 清除客户区；
- **WM_MOUSEMOVE** 如果按下左键，那么 CONNECT 就在客户区中的鼠标位置处绘制一个黑点，并保存该坐标；
- **WM_LBUTTONUP** CONNECT 把客户区中绘制的点同其他每个点连接起来。有时会产生一个漂亮的图形，而有时则会是黑乎乎的一团糟(见图 7-1)。

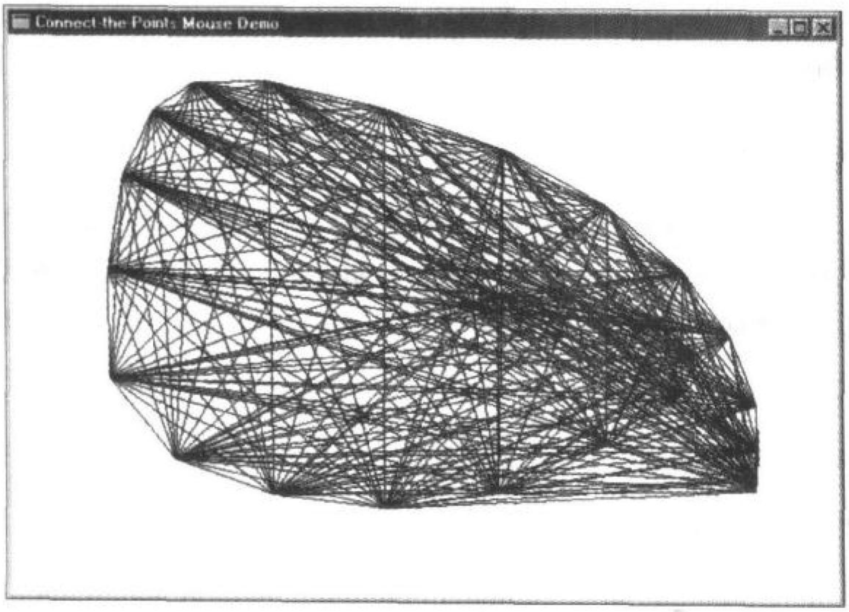


图 7-1 CONNECT 的屏幕显示

CONNECT的使用方法：把鼠标光标移动到客户区中，按下左键，移动一下位置，释放左键。对几个构成曲线的点，CONNECT能处理得很好，方法是按住左键，快速移动鼠标，这样就可以绘制出该曲线图案。

CONNECT使用了三个简单的图形设备接口（GDI）函数，在第五章讨论过这些函数。当鼠标左键按下时，SetPixel为每个WM_MOUSEMOVE消息绘制一个黑像素（对高分辨率的显示器，像素几乎看不见）。画直线需要MoveToEx和LineTo函数。

如果您在释放鼠标键之前把鼠标光标移到客户区之外，那么CONNECT就不会连接这些点，因为它没有收到WM_LBUTTONUP消息。如果把鼠标移回客户区内并按下左键，那么CONNECT将清除客户区。如果想在客户区外释放左键后还继续进行画图，那么可以在客户区外按下鼠标再移回客户区中。

CONNECT最多可以保存1000个点。设点数为P，则CONNECT画的线数就等于P×（P-1）/2。如果有1000个点，则要绘制50万条直线，大约需要几分钟才能画完（时间的长短取决于您的硬件）。由于Windows 98是一种抢占式多任务环境，因此您可以在这一段时间切换到别的程序中。但是，当程序正在忙的时候，您将无法对CONNECT程序做任何事（诸如移动或者缩放等）。在第二十章中，我们将讨论解决这一问题的方法。

因为CONNECT可能会花一些时间来绘制直线，因此在处理WM_PAINT消息时它将切换到沙漏光标，然后再恢复原状。这要求使用两个现有光标来调用SetCursor。CONNECT还调用两次ShowCursor，一次用TRUE参数，另一次用FALSE参数。我将在7.4.3节“使用键盘仿真鼠标”中更详细地讨论这些调用。

有时，我们使用“跟踪”这个词代表程序处理鼠标移动的方法。但是，跟踪并不意味着程序在窗口过程中的某个循环里不断跟随鼠标在显示器上的运动。实际上，窗口过程处理每条鼠标消息，然后迅速退出。

7.2.2 处理Shift键

当CONNECT接收到一个WM_MOUSEMOVE消息时，它把wParam和MK_LBUTTON进行按位与（AND）运算，来确定是否按下了左键。wParam也可以用于确定Shift键的状态。例如，如果处理必须依赖于Shift和Ctrl键的状态，那么您可以使用如下所示的方法：

```
if (wParam & MK_SHIFT)
{
     if (wParam & MK_CONTROL)
     {
          [Shift and Ctrl keys are down]
     }
     else
     {
          [Shift key is down]
```

```
        }
    }
    else
    {
        if (wParam & MK_CONTROL)
        {
            [Ctrl key is down]
        }
        else
        {
            [neither Shift nor Ctrl key is down]
        }
    }
```

如果您想在程序中同时使用左右键，同时如果还希望只有单键鼠标的用户也能使用您的程序，那么您可以这样来编写代码：Shift 与左键的组合使用等效于右键。在这种情况下，对鼠标键的处理可以采用如下所示的方法：

```
case WM_LBUTTONDOWN:
    if (!(wParam & MK_SHIFT))
    {
        [left button logic]
        return 0 ;
    }
                    // Fall through
case WM_RBUTTONDOWN:
    [right button logic]
    return 0 ;
```

Window 函数 GetKeyState（在第六章中介绍过）可以使用虚拟键码 VK_LBUTTON、VK_RBUTTON、VK_MBUTTON、VK_SHIFT 和 VK_CONTROL 来返回鼠标键与 shift 键的状态。如果 GetKeyState 返回负值，则说明按下了鼠标键或者 shift 键。因为 GetKeyState 返回当前正在处理的鼠标键或者 shift 键的状态，所以全部状态信息都与相应的消息是同步的。但是，正如不能把 GetKeyState 用于尚未按下的键一样，您也不能为尚未按下的鼠标键调用 GetKeyState。不要这样做：

```
while (GetKeyState (VK_LBUTTON) >= 0) ; // WRONG !!!
```

只有在您调用 GetKeyState 期间处理消息时，如果左键已经按下，才会报告键已经按下的消息。

7.2.3 双击鼠标键

双击鼠标键是指在短时间内单击两次。要确定为双击，这两次单击必须发生在其相互

的物理位置十分接近的状况下(默认时范围是一个平均系统字体字符的宽,半个字符的高),并且发生在指定的时间间隔(称为"双击速度")内。您可以在"控制面板"中改变该时间间隔。

如果希望窗口过程能够收到双击键的鼠标消息,那么在调用 RegisterClass 初始化窗口类结构中时,必须在窗口风格中包含 CS_DBLCLKS 标识符:

```
wndclass.style = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS ;
```

如果在窗口风格中未包含 CS_DBLCLKS,而用户在短时间内双击了鼠标键,那么窗口过程接收到下面这些消息:

WM_LBUTTONDOWN

WM_LBUTTONUP

WM_LBUTTONDOWN

WM_LBUTTONUP

窗口过程可能在这些键消息之前还收到了其他消息。如果您想实现自己的双击逻辑,那么您可以使用 Windows 函数 GetMessageTime 获取 WM_LBUTTONDOWN 消息之间的相对时间。第八章将更详细地讨论这个函数。

如果窗口类风格中包含了 CS_DBLCLKS,那么双击鼠标右键时窗口过程将接收到如下消息:

WM_LBUTTONDOWN

WM_LBUTTONUP

WM_LBUTTONDBLCLK

WM_LBUTTONUP

WM_LBUTTONDBLCLK 消息简单地替换了第二个 WM_LBUTTONDOWN 消息。

如果双击中的第一次单击操作完成单个单击的操作功能,那么处理双击消息是很容易的。第二次单击操作(WM_LBUTTONDBLCLK 消息)则完成第一次单击操作以外的事情。例如,看看"Windows 资源管理器"中是如何用鼠标来操作文件列表的。单击一次将选中文件,"Windows 资源管理器"用反转视频显示条加亮文件。双击则实现两个功能:第一次单击操作如单个单击那样选中文件;第二次单击操作则指示"Windows 资源管理器"打开该文件。这是一个相当简单的逻辑。如果双击中的第一次单击操作不执行单个单击功能,那么鼠标处理逻辑可能会变得非常复杂。

7.3 非客户区鼠标消息

在窗口的客户区内移动或按下鼠标键时,将产生10个消息。如果鼠标在窗口的客户区之外但还在窗口内,Windows就给窗口过程发送一个“非客户区”鼠标消息。窗口非客户区包括标题栏、菜单和窗口滚动条。

通常,您不需要处理非客户区鼠标消息,而将这些消息传给DefWindowProc,从而使Windows执行系统功能。就这方面来说,非客户区鼠标消息类似于系统键盘消息WM_SYSKEYDOWN、WM_SYSKEYUP和WM_SYSCHAR。

非客户区鼠标消息几乎完全与客户区鼠标消息平行出现。消息中含有字母“NC”以表示是非客户区消息。如果鼠标在窗口的非客户区中移动,那么窗口过程接收到WM_NCMOUSEMOVE消息。鼠标键产生如表7-2所示的消息。

表7-2

键	按下	释放	按下(双击)
左	WM_NCLBUTTONDOWN	WM_NCLBUTTONUP	WM_NCLBUTTONDBLCLK
中	WM_NCMBUTTONDOWN	WM_NCMBUTTONUP	WM_NCMBUTTONDBLCLK
右	WM_NCRBUTTONDOWN	WM_NCRBUTTONUP	WM_NCRBUTTONDBLCLK

非客户区鼠标消息的wParam和lParam参数与客户区鼠标消息的wParam和lParam参数有一定的差别。wParam参数指明移动或者单击鼠标键的非客户区位置。它设置为WINUSER.H中定义的以HT开头的标识符之一(HT表示“命中测试”)。

lParam参数包含低位字的x坐标和高位字的y坐标,但是,它们都是屏幕坐标,而不是像在客户区鼠标消息中那样指的是客户区坐标。对于屏幕坐标,显示器左上角的x值和y值都为0。当往右移时x的值增加,往下移时y的值增加(见下页图7-2)。

您可以用以下这两个Windows函数将屏幕坐标转换为客户区坐标,或者将客户区坐标转换为屏幕坐标:

```
ScreenToClient (hwnd, &pt) ;
ClientToScreen (hwnd, &pt) ;
```

这里pt是POINT结构。这两个函数转换了保存在结构中的值,而且没有保留以前的值。注意,如果屏幕坐标点在窗口客户区的上面或者左边,客户区坐标x或y值就是负值。

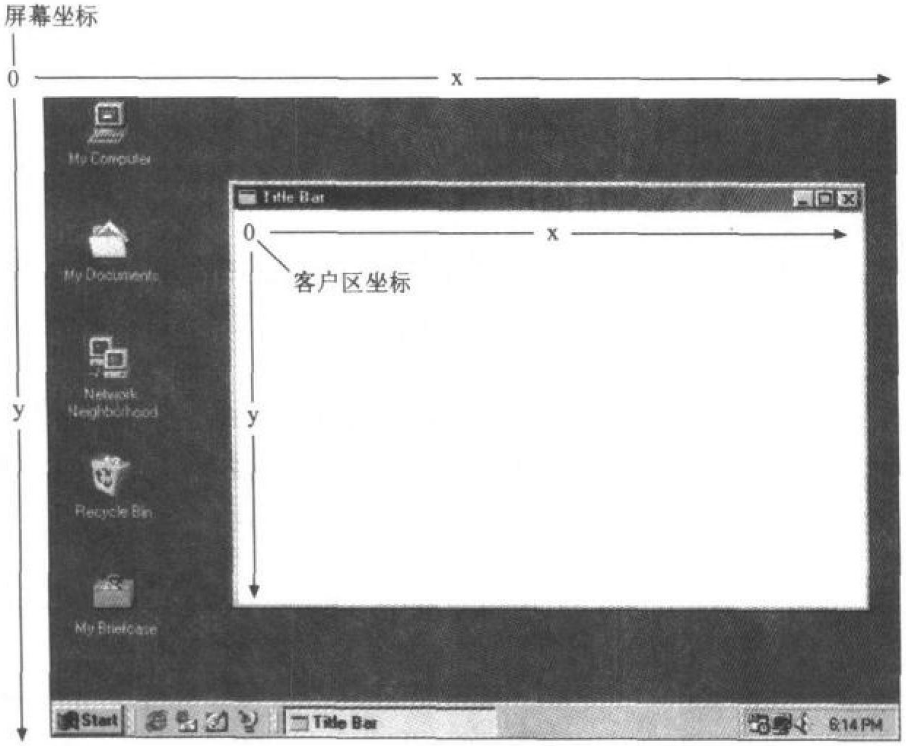


图 7-2 屏幕坐标与客户区坐标

7.3.1 命中测试消息

如果您数一下，就可以知道我们已经介绍了 21 个鼠标消息中的 20 个，最后一个消息是 WM_NCHITTEST，它代表“非客户区命中测试”。此消息优先于所有其他的客户区和非客户区鼠标消息。lParam 参数含有鼠标位置的 x 和 y 屏幕坐标，wParam 参数没有用。

Windows 应用程序通常把这个消息传送给 DefWindowProc，然后 Windows 用 WM_NCHITTEST 消息产生基于鼠标位置的所有其他鼠标消息。对于非客户区鼠标消息，当处理 WM_NCHITTEST 时，从 DefWindowProc 返回的值将成为鼠标消息中的 wParam 参数，这个值可以是任意非客户区鼠标消息的 wParam 值再加上以下内容：

HTCLIENT	客户区
HTNOWHERE	不在窗口中
HTTRANSPARENT	窗口由另一个窗口覆盖
HTERROR	使 DefWindowProc 产生蜂鸣声

如果 DefWindowProc 在其处理 WM_NCHITTEST 消息后返回 HTCLIENT，那么 Windows 将把屏幕坐标转换为客户区坐标并产生客户区鼠标消息。

如果您还记得我们如何通过捕获 WM_SYSKEYDOWN 消息来禁止所有的系统键盘

功能，那么您可能想我们可否通过捕获鼠标消息完成类似的事情。完全可以！只要您在窗口过程中包含以下几条语句：

```
case WM_NCHITTEST:
    return (LRESULT) HTNOWHERE ;
```

就可以有效地禁止您窗口中的所有客户区和非客户区鼠标消息。这样一来，当鼠标在您的窗口（包括系统菜单图标、缩放按钮以及关闭按钮）中时，鼠标键将失效。

7.3.2 从消息产生消息

Windows 用 WM_NCHITTEST 消息产生所有其他鼠标消息，这种由消息引出其他消息的思想在 Windows 中是很普遍的。让我们来举个例子。您知道，如果您在一个 Windows 程序的系统菜单图标上双击一下，那么程序将会终止。双击产生一系列的 WM_NCHITTEST 消息。由于鼠标定位在系统菜单图标上，因此 DefWindowProc 将返回 HTSYSMENU 的值，并且 Windows 把 wParam 等于 HTSYSMENU 的 WM_NCLBUTTONDBLCLK 消息放在消息队列中。

窗口过程通常把鼠标消息传递给 DefWindowProc，当 DefWindowProc 接收到 wParam 参数等于 HTSYSMENU 的 WM_NCLBUTTONDBLCLK 消息时，它就把 wParam 参数等于 SC_CLOSE 的 WM_SYSCOMMAND 消息放入消息队列中（这个 WM_SYSCOMMAND 消息是在用户从系统菜单中选择 Close 时产生的）。同样，窗口过程也把这个消息传给 DefWindowProc。DefWindowProc 通过给窗口过程发送 WM_CLOSE 消息来处理该消息。

如果一个程序在终止之前要求来自用户的确认，那么窗口过程就需要捕获 WM_CLOSE，否则，DefWindowProc 调用 DestroyWindow 函数来处理 WM_CLOSE。除了其他处理，DestroyWindow 还给窗口过程发送一个 WM_DESTROY 消息。窗口过程通常用下列代码来处理 WM_DESTROY 消息：

```
case WM_DESTROY:
    PostQuitMessage (0) ;
    return 0 ;
```

PostQuitMessage 使 Windows 把 WM_QUIT 消息放入消息队列中，此消息永远不会到达窗口过程，因为它使 GetMessage 返回 0，终止消息循环，从而也终止了程序。

7.4 程序中的命中测试

在前面讨论了“Windows 资源管理器”如何响应鼠标的单击和双击。显然，程序（或者“Windows 资源管理器”刚好使用列表视图）必须确定用户鼠标所指向的是哪一个文件。

这叫做“命中测试”。正如 DefWindowProc 在处理 WM_NCHITTEST 消息时做一些命中测试一样，窗口过程经常必须在客户区中进行一些命中测试。一般来讲，命中测试中会使用 x 和 y 坐标值，它们由传到窗口过程的鼠标消息的 lParam 参数给出。

7.4.1 一个假想的例子

有这样一个例子。假设您的程序需要显示几列按字母排列的文件。通常，由于要做全部的命中测试，所以可使用列表视图。但我们假设您由于某种原因而不能使用。这时就需要自己来做了。让我们假定文件名保存在被称为 szFileNames 的指向已排序的字符串指针数组中。

让我们也假定文件列表开始于客户区的顶端，客户区为 cxClient 像素宽，cyClient 像素高，每列为 cxColWidth 像素宽，每个字符高度为 cyChar 像素高。那么每栏可填充的文件数就是：

```
iNumInCol = cyClient / cyChar ;
```

接收到一个鼠标单击消息后，您就能从 lParam 获得 cxMouse 和 cyMouse 坐标。然后可以用下面的公式来计算用户所指的是哪一列的文件名：

```
iColumn = cxMouse / cxColWidth ;
```

相对于列顶端的文件名位置为

```
iFromTop = cyMouse / cyChar ;
```

现在您就可以计算 szFileNames 数组的下标：

```
iIndex = iColumn * iNumInCol + iFromTop ;
```

如果 iIndex 超过了数组中的文件数，则表示用户是在显示器的空白区域内按鼠标键。

在许多情况下，命中测试要比本例更加复杂。在显示一幅包含许多小图形的图像时，您必须决定要显示的每个小图形的坐标。在命中计算中，您必须从坐标找到对象。但这将在使用不确定字体大小的字处理程序中变得非常凌乱，因为您必须找到字符在串中的位置。

7.4.2 示例程序

程序 7-2 所示的 CHECKER1 程序演示了一些简单的命中测试，此程序把客户区分为 5×5的 25 个矩形。如果您在某个矩形中按下鼠标键，那么在该矩形中将出现一个“X”。如果您再按一次，那么“X”将被删除。

程序 7-2 CHECKER1 程序

CHECKER1.C

```
/*-------------------------------------------------
   CHECKER1.C——Mouse Hit-Test Demo Program No. 1
               (c) Charles Petzold, 1998
  -------------------------------------------------*/

#include <windows.h>

#define DIVISIONS 5

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("Checker1") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }
     hwnd = CreateWindow (szAppName, TEXT ("Checker1 Mouse Hit-Test Demo"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;
```

```
     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static BOOL  fState[DIVISIONS][DIVISIONS] ;
     static int   cxBlock, cyBlock ;
     HDC          hdc ;
     int          x, y ;
     PAINTSTRUCT  ps ;
     RECT         rect ;

     switch (message)
     {
     case WM_SIZE :
          cxBlock = LOWORD (lParam) / DIVISIONS ;
          cyBlock = HIWORD (lParam) / DIVISIONS ;
          return 0 ;

     case WM_LBUTTONDOWN :
          x = LOWORD (lParam) / cxBlock ;
          y = HIWORD (lParam) / cyBlock ;

          if (x < DIVISIONS && y < DIVISIONS)
          {
               fState [x][y] ^= 1 ;

               rect.left   = x * cxBlock ;
               rect.top    = y * cyBlock ;
               rect.right  = (x + 1) * cxBlock ;
               rect.bottom = (y + 1) * cyBlock ;

               InvalidateRect (hwnd, &rect, FALSE) ;
          }
          else
               MessageBeep (0) ;
          return 0 ;

     case WM_PAINT :
          hdc = BeginPaint (hwnd, &ps) ;
```

```
          for (x = 0 ; x < DIVISIONS ; x++)
          for (y = 0 ; y < DIVISIONS ; y++)
          {
               Rectangle (hdc, x * cxBlock, y * cyBlock,
                         (x + 1) * cxBlock, (y + 1) * cyBlock) ;

               if (fState [x][y])
               {
                    MoveToEx (hdc, x     * cxBlock, y     * cyBlock, NULL) ;
                    LineTo   (hdc, (x+1) * cxBlock, (y+1) * cyBlock) ;
                    MoveToEx (hdc, x     * cxBlock, (y+1) * cyBlock, NULL) ;
                    LineTo   (hdc, (x+1) * cxBlock, y     * cyBlock) ;
               }
          }
          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

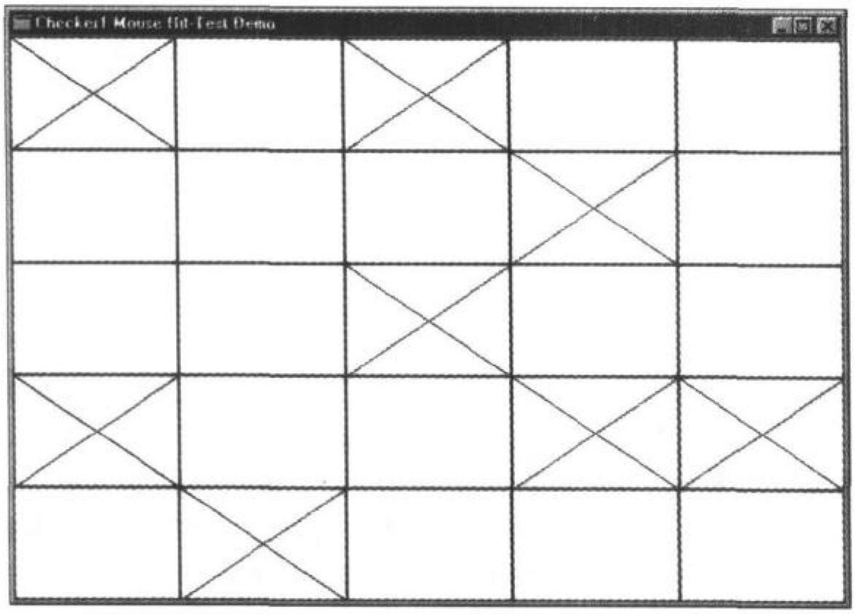


图 7-3 CHECKER1 的屏幕显示

图 7-3 是 CHECKER1 的显示。程序画的 25 个矩形,其宽度均相同,并且其高度也均相同。这些宽度和高度保存在 cxBlock 和 cyBlock 中,当客户区大小发生改变时将重

新对这些值进行计算。WM_LBUTTONDOWN 处理过程使用鼠标坐标来确定在哪个矩形中按下了键，它在 fState 数组中标志当前矩形的状态，并使该矩形区域失效，从而产生 WM_PAINT 消息。

如果客户区的宽度和高度不能被 5 整除，那么在客户区的左边和下边将有一小条不能被矩形所覆盖。对出错情况，CHECKER1 通过调用 MessageBeep 响应此区域中的鼠标键操作。

当 CHECKER1 收到 WM_PAINT 消息时，它通过 GDI 的 Rectangle 函数来重新绘制客户区。如果设置了 fState 值，那么 CHECKER1 将使用 MoveToEx 和 LineTo 函数来绘制两条直线。在处理 WM_PAINT 期间，CHECKER1 在重新绘制之前并不检查每个矩形区域的有效性，尽管它可以这样做。检查有效性的一种方法是在循环中为每个矩形块构造 RECT 结构（使用 WM_LBUTTONDOWN 逻辑中相同的公式），并使用 IntersectRect 函数检查它是否同无效矩形（ps.rcPaint）相交。

7.4.3 使用键盘仿真鼠标

CHECKER1 只能在装有鼠标时才可运行。下面我们在程序中加入键盘接口，就如同第六章中对 SYSMETS 程序所做的那样。不过，在一个使用鼠标光标的程序中加入键盘接口要求我们必须处理鼠标光标的移动和显示问题。

即使没有安装鼠标，Windows 仍然可以显示一个鼠标光标。Windows 为这个光标保存了一个“显示计数”。如果安装了鼠标，显示计数被初始化为 0；否则，显示计数被初始化为 -1。只有在显示计数非负时才显示鼠标光标。要增加显示计数，您可以调用

```
ShowCursor (TRUE) ;
```

要减少显示计数，可以调用

```
ShowCursor (FALSE) ;
```

在使用 ShowCursor 之前，不需要确定是否安装了鼠标。如果您想显示鼠标光标，而不管鼠标存在与否，那么只需调用 ShowCursor 来增加显示计数。增加一次显示计数之后，如果没有安装鼠标则减少它将隐藏光标；如果安装了鼠标，则保留其显示。

即使没有安装鼠标，Windows 也保留了鼠标的当前位置。如果没有安装鼠标，而您又显示鼠标光标，则它会出现在显示器的任何部分并将停留在那个位置，直到您确实移动了它。要获得光标的位置，可以调用：

```
GetCursorPos (&pt) ;
```

其中 pt 是 POINT 结构。函数使用鼠标的 x 和 y 坐标来填充 POINT 域。要设置光标位置，可以使用

```
SetCursorPos (x, y) ;
```

在这两种情况下，x 和 y 都是屏幕坐标，而不是客户区坐标（这是显然的，因为这些函数没有要求 hwnd 参数）。前面已经提到过，调用 ScreenToClient 和 ClientToScreen 就能做到屏幕坐标与客户坐标的相互转换。

如果您在处理鼠标消息并转换客户区坐标时调用 GetCursorPos，这些坐标可能与鼠标消息的 lParam 参数中的坐标稍微有些不同。从 GetCursorPos 返回的坐标表明鼠标的当前位置。lParam 中的坐标是产生消息时鼠标的位置。

您或许想编写一个键盘逻辑：使用键盘方向键来移动鼠标光标，使用 Spacebar 和 Enter 键来模拟鼠标键。您肯定不希望每次击键只是将鼠标光标移动一个像素，这样做，当把鼠标光标从显示器的一边移动到另一边时，会使用户按住一个方向键的时间很长。

如果您需要实现鼠标光标的键盘接口并保持光标的精确定位能力，那么您可以采用下面的方式来处理击键消息：当按下方向键时，鼠标光标开始移动较慢，但随后会加快。您也许还记得 WM_KEYDOWN 消息中的 lParam 参数标志着击键消息是否是重复活动的结果，这就是此参数的一个重要应用。

7.4.4 在 CHECKER 中添加键盘接口

程序 7-3 所示的 CHECKER2 程序，除了包括键盘接口外，同 CHECKER1 是一样的，您可以使用左、右、上和下方向键在 25 个矩形之间移动光标。Home 键把光标移动到矩形的左上角，End 键把光标移动到矩形的右下角。Spacebar 和 Enter 键都能切换 X 标记。

程序 7-3 CHECKER2 程序

CHECKER2.C

```
/*-------------------------------------------
     CHECKER2.C——Mouse Hit-Test Demo Program No. 2
                 (c) Charles Petzold, 1998
  -------------------------------------------*/

#include <windows.h>

#define DIVISIONS 5

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
```

```
     static TCHAR szAppName[] = TEXT ("Checker2") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }
     hwnd = CreateWindow (szAppName, TEXT ("Checker2 Mouse Hit-Test Demo"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static BOOL  fState[DIVISIONS][DIVISIONS] ;
     static int   cxBlock, cyBlock ;
     HDC          hdc ;
     int          x, y ;
     PAINTSTRUCT  ps ;
     POINT        point ;
```

```
RECT        rect ;

switch (message)
{
case WM_SIZE :
    cxBlock = LOWORD (lParam) / DIVISIONS ;
    cyBlock = HIWORD (lParam) / DIVISIONS ;
    return 0 ;

case WM_SETFOCUS :
    ShowCursor (TRUE) ;
    return 0 ;

case WM_KILLFOCUS :
    ShowCursor (FALSE) ;
    return 0 ;

case WM_KEYDOWN :
    GetCursorPos (&point) ;
    ScreenToClient (hwnd, &point) ;
    x = max (0, min (DIVISIONS - 1, point.x / cxBlock)) ;
    y = max (0, min (DIVISIONS - 1, point.y / cyBlock)) ;

    switch (wParam)
    {
    case VK_UP :
        y-- ;
        break ;

    case VK_DOWN :
        y++ ;
        break ;

    case VK_LEFT :
        x-- ;
        break ;

    case VK_RIGHT :
        x++ ;
        break ;

    case VK_HOME :
        x = y = 0 ;
        break ;

    case VK_END :
        x = y = DIVISIONS - 1 ;
```

```
                    break ;

               case VK_RETURN :
               case VK_SPACE :
                    SendMessage (hwnd, WM_LBUTTONDOWN, MK_LBUTTON,
                                 MAKELONG (x * cxBlock, y * cyBlock)) ;
                    break ;
               }
               x = (x + DIVISIONS) % DIVISIONS ;
               y = (y + DIVISIONS) % DIVISIONS ;

               point.x = x * cxBlock + cxBlock / 2 ;
               point.y = y * cyBlock + cyBlock / 2 ;

               ClientToScreen (hwnd, &point) ;
               SetCursorPos (point.x, point.y) ;
               return 0 ;
          case WM_LBUTTONDOWN :
               x = LOWORD (lParam) / cxBlock ;
               y = HIWORD (lParam) / cyBlock ;

               if (x < DIVISIONS && y < DIVISIONS)
               {
                    fState[x][y] ^= 1 ;

                    rect.left   = x * cxBlock ;
                    rect.top    = y * cyBlock ;
                    rect.right  = (x + 1) * cxBlock ;
                    rect.bottom = (y + 1) * cyBlock ;

                    InvalidateRect (hwnd, &rect, FALSE) ;
               }
               else
                    MessageBeep (0) ;
               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               for (x = 0 ; x < DIVISIONS ; x++)
               for (y = 0 ; y < DIVISIONS ; y++)
               {
                    Rectangle (hdc, x * cxBlock, y * cyBlock,
                              (x + 1) * cxBlock, (y + 1) * cyBlock) ;

                    if (fState [x][y])
                    {
                         MoveToEx (hdc, x     * cxBlock, y     * cyBlock, NULL) ;
```

```
                    LineTo   (hdc, (x+1) * cxBlock, (y+1) * cyBlock) ;
                    MoveToEx (hdc, x     * cxBlock, (y+1) * cyBlock, NULL) ;
                    LineTo   (hdc, (x+1) * cxBlock, y     * cyBlock) ;
               }
          }
          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

CHECKER2 中的 WM_KEYDOWN 逻辑决定光标的位置(用 GetCursorPos),把屏幕坐标转换为客户区坐标(用 ScreenToClient),并用矩形块的宽度和高度来除这个坐标。这会产生指出 5×5 数组中矩形的位置的 x 和 y 值。当按下一个键时,鼠标光标可能在或不在客户区中,从而 x 和 y 必须经 min 和 max 宏处理以保证它们的范围是 0 到 4。

对方向键,CHECKER2 近似地增加或减少 x 和 y。如果是 Enter 键或 Spacebar 键,那么 CHECKER2 使用 SendMessage 把 WM_LBUTTONDOWN 消息发送给它自身。这种技术类似于在第六章 SYSMETS 程序中把键盘接口加到窗口滚动条时所使用的方法。WM_KEYDOWN 逻辑是通过计算指向矩形中心的客户区坐标,再用 ClientToScreen 转换成屏幕坐标,然后用 SetCursorPos 设置光标位置来实现的。

7.4.5 将子窗口用于命中测试

有些程序(例如,Windows 的“画图”程序),把客户区划分为几个小的逻辑区域。“画图”程序在其左边有一个基于图标的工具菜单区,在底部有颜色菜单区。在这两个区命中测试的时候,“画图”必须在用户选中菜单项之前记住菜单的位置。

不过,也可能不需要这么做。实际上,Paint 通过使用子窗口简化了菜单的绘制和命中测试。子窗口把整个矩形区域划分为几个更小的矩形区,每个子窗口有自己的窗口句柄、窗口过程和客户区,每个窗口过程接收只适用于它的子窗口的鼠标消息。鼠标消息中的 lParam 参数含有相当于该子窗口客户区左上角的坐标,而不是其父窗口(那是“画图”的主应用程序窗口)客户区左上角的坐标。

这种方法使用的子窗口有助于程序的结构化和模块化。如果子窗口使用不同的窗口类,那么每个子窗口都有它自己的窗口过程。不同的窗口也可以定义不同的背景颜色和不同的默认光标。在第九章中,我们将看到“子窗口控制”——滚动条、按钮和编辑框等预先定义的子窗口。现在,我们说明在 CHECKER 程序中是如何使用子窗口的。

7.4.6 CHECKER 中的子窗口

程序 7-4 给出了 CHECKER3,这一版本创建了 25 个处理鼠标单击的子窗口。它没有键盘接口,但是可以按本章中随后介绍的 CHECKER4 所示例的方法添加。

程序 7-4 CHECKER3 程序

CHECKER3.C

```
/*-------------------------------------------------
   CHECKER3.C -- Mouse Hit-Test Demo Program No. 3
                 (c) Charles Petzold, 1998
  -------------------------------------------------*/

#include <windows.h>

#define DIVISIONS 5

LRESULT CALLBACK WndProc   (HWND, UINT, WPARAM, LPARAM) ;
LRESULT CALLBACK ChildWndProc (HWND, UINT, WPARAM, LPARAM) ;

TCHAR szChildClass[] = TEXT ("Checker3_Child") ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("Checker3") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
```

```
          return 0 ;
     }

     wndclass.lpfnWndProc    = ChildWndProc ;
     wndclass.cbWndExtra     = sizeof (long) ;
     wndclass.hIcon          = NULL ;
     wndclass.lpszClassName  = szChildClass ;

     RegisterClass (&wndclass) ;

     hwnd = CreateWindow (szAppName, TEXT ("Checker3 Mouse Hit-Test Demo"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static HWND hwndChild[DIVISIONS][DIVISIONS] ;
     int         cxBlock, cyBlock, x, y ;

     switch (message)
     {
     case WM_CREATE :
          for (x = 0 ; x < DIVISIONS ; x++)
               for (y = 0 ; y < DIVISIONS ; y++)
                    hwndChild[x][y] = CreateWindow (szChildClass, NULL,
                              WS_CHILDWINDOW | WS_VISIBLE,
                              0, 0, 0, 0,
                              hwnd, (HMENU) (y << 8 | x),
                              (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE),
                              NULL) ;
          return 0 ;

     case WM_SIZE :
          cxBlock = LOWORD (lParam) / DIVISIONS ;
          cyBlock = HIWORD (lParam) / DIVISIONS ;
```

```
          for (x = 0 ; x < DIVISIONS ; x++)
               for (y = 0 ; y < DIVISIONS ; y++)
                    MoveWindow (hwndChild[x][y],
                                x * cxBlock, y * cyBlock,
                                cxBlock, cyBlock, TRUE) ;
          return 0 ;

     case WM_LBUTTONDOWN :
          MessageBeep (0) ;
          return 0 ;

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}

LRESULT CALLBACK ChildWndProc (HWND hwnd, UINT message,
                               WPARAM wParam, LPARAM lParam)
{
     HDC         hdc ;
     PAINTSTRUCT ps ;
     RECT        rect ;

     switch (message)
     {
     case WM_CREATE :
          SetWindowLong (hwnd, 0, 0) ;       // on/off flag
          return 0 ;

     case WM_LBUTTONDOWN :
          SetWindowLong (hwnd, 0, 1 ^ GetWindowLong (hwnd, 0)) ;
          InvalidateRect (hwnd, NULL, FALSE) ;
          return 0 ;

     case WM_PAINT :
          hdc = BeginPaint (hwnd, &ps) ;

          GetClientRect (hwnd, &rect) ;
          Rectangle (hdc, 0, 0, rect.right, rect.bottom) ;

          if (GetWindowLong (hwnd, 0))
          {
               MoveToEx (hdc, 0,          0, NULL) ;
               LineTo   (hdc, rect.right, rect.bottom) ;
               MoveToEx (hdc, 0,          rect.bottom, NULL) ;
               LineTo   (hdc, rect.right, 0) ;
```

```
        }

        EndPaint (hwnd, &ps) ;
        return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

CHECKER3 有名为 WndProc 和 ChildWndProc 的两个窗口过程。WndProc 还是主(或父)窗口的窗口过程。ChildWndProc 是针对 25 个子窗口的窗口过程。这两个窗口过程都必须定义为 CALLBACK 函数。

因为窗口过程与特定的窗口类结构相关联,该窗口类结构由 Windows 调用 RegisterClass 函数来注册,CHECKER3 需要两个窗口类。第一个窗口类用于主窗口,名为"Checker3"。第二个窗口类名为"Checker3_Child"。当然,您不必选择像这样有意义的名称。

CHECKER3 在 WinMain 函数中注册了这两个窗口类。注册完常规的窗口类之后,CHECKER3 只是简单地重新使用 wndclass 结构中的大多数的域来注册 Checker3_Child 类。无论如何,4 个域根据子窗口类而设置为不同的值:

- lpfnWndProc 域设置为 ChildWndProc,子窗口类的窗口过程。
- cbWndExtra 字段设置为 4 字节,或者更确切地,用 sizeof (long)。该域告诉 Windows 在其为基于此窗口类的每个窗口保留的内部结构中预留了 4 字节额外的空间。您能使用此空间来保存每个窗口的、可能有所不同的信息。
- 因为像 CHECKER3 中的子窗口不需要图标,所以 hIcon 域设置为 NULL 。
- pszClassName 域设置为"Checker3_Child",是类的名称。

WinMain 中的 CreateWindow 调用创建基于 Checker3 类的主窗口。这是很正常的。然而,当 WndProc 接收到一个 WM_CREATE 消息时,它将调用 CreateWindow 25 次,以创建 25 个基于 Checker3_Child 类的子窗口。下页表 7-3 包含了 WinMain 中的 CreateWindow 调用与创建 25 个子窗口的 WndProc 中的 CreateWindow 调用的参数间的比较。

通常情况下,子窗口要求有关位置和大小的参数,但是在 CHECKER3 中,子窗口稍后会在 WndProc 中确定位置和大小。因为它是父窗口,所以对于主窗口,该父窗口句柄是 NULL。当使用 CreateWindow 调用来创建一个子窗口时,就需要父窗口句柄了。

主窗口没有菜单,因此其参数为 NULL。对于子窗口,相同位置的参数被称为"子 ID"(或"子窗口 ID")。这是唯一标识子窗口的数值。像我们在第十一章所看到的一样,在处理对话框的子窗口控制时,子 ID 就会显得更为重要。对于 CHECKER3 来说,我只是简单地将子 ID 设置为一个数,该数是每个子窗口在 5×5 的主窗口中的 x 和 y 位置的组合。

CreateWindow 函数需要一个实例句柄。在 WinMain 中,实例句柄可以很容易实现,因为它是 WinMain 的一个参数。在创建子窗口时,CHECKER3 必须用 GetWindowLong 来从 Windows 为窗口保留的结构中获取 hInstance 值。(相对于 GetWindowLong,我也能将 hInstance 的值保存到全局变量,并直接使用它。)

表 7-3

参数	主窗口	子窗口
窗口类	"Checker3"	"Checker3_Child"
窗口标题	"Checker3..."	NULL
窗口风格	WS_OVERLAPPEDWINDOW	WS_CHILDWINDOW | WS_VISIBLE
水平位置	CW_USEDEFAULT	0
垂直位置	CW_USEDEFAULT	0
宽度	CW_USEDEFAULT	0
高度	CW_USEDEFAULT	0
父窗口句柄	NULL	hwnd
菜单句柄/子 ID	NULL	(HMENU) (y << 8 | x)
实例句柄	hInstance	(HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE)
额外参数	NULL	NULL

每一个子窗口都在 hwndChild 数组中保存了不同的窗口句柄。当 WndProc 接收到了一条 WM_SIZE 消息后,它将为这 25 个子窗口调用 MoveWindow。MoveWindow 的参数表明子窗口左上角相对于父窗口客户区的坐标、子窗口的宽度和高度,以及子窗口是否需要重画。

现在让我们看一下 ChildWndProc。此窗口过程为所有这 25 个子窗口处理消息。ChildWndProc 的 hwnd 参数是子窗口接收消息的句柄。当 ChildWndProc 处理 WM_CREATE 消息时(因为有 25 个子窗口,所以要发生 25 次),它用 SetWindowWord 在窗口结构保留的额外区域中存储一个 0 值。(通过在定义窗口类时使用的 cbWndExtra 来调用我们保留的空间。)ChildWndProc 用此值来恢复当前矩形的状态(X 或没有 X)。在子窗口中单击时,WM_LBUTTONDOWN 逻辑简单地修改这个整数值(从 0 到 1 或从 1 到 0),并使整个子窗口无效。此区域是单击的矩形。WM_PAINT 的处理很普通,因为它所绘制的矩形与客户区一样大。

因为 CHECKER3 的 C 源代码文件和 .EXE 文件比 CHECKER1 的大(更不用说程序的说明了),我不是要说服您 CHECKER3 比 CHECKER1 更简单。但请注意,我们没有做任何的鼠标命中测试!如果 CHECKER3 中的一个子窗口将获得命中窗口的 WM_LBUTTONDOWN 消息,这就是所有要说明的。

7.4.7 子窗口和键盘

为 CHECKER3 添加键盘接口就像 CHECKER 系列最后一步的逻辑。但在这样做的时候,可能需要适当的路径。在 CHECKER2 中,鼠标光标的位置决定按下 Spacebar 键时哪个区域将获得标记符号。当我们处理子窗口时,我们能从对话框功能中获得提示。在对话框中,带有闪烁的字符或点划的矩形的子窗口表明它有输入焦点(当然也可以用键盘进行定位)。

我们不再复制 Windows 内部已有的所有对话框逻辑,我只是要告诉您如何在应用程序中仿真对话框。研究过程中,您会发现这样一件事:父窗口和子窗口可能要共享键盘消息处理。按下 Spacebar 键和 Enter 键时,子窗口将锁定复选标记。按下方向键时,父窗口将在子窗口之间移动输入焦点。实际上,当您在子窗口上单击时,情况有些复杂,这时是父窗口而不是子窗口获得输入焦点。

CHECKER4.C 如程序 7-5 所示。

程序 7-5 CHECKER4 程序

CHECKER4.C

```
/*------------------------------------------
   CHECKER4.C——Mouse Hit-Test Demo Program No. 4
              (c) Charles Petzold, 1998
  ------------------------------------------*/

#include <windows.h>
#define DIVISIONS 5

LRESULT CALLBACK WndProc   (HWND, UINT, WPARAM, LPARAM) ;
LRESULT CALLBACK ChildWndProc (HWND, UINT, WPARAM, LPARAM) ;
int    idFocus = 0 ;
TCHAR szChildClass[] = TEXT ("Checker4_Child") ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("Checker4") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
```

```
          wndclass.cbWndExtra     = 0 ;
          wndclass.hInstance      = hInstance ;
          wndclass.hIcon          = LoadIcon (NULL, IDI_APPLICATION) ;
          wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
          wndclass.hbrBackground  = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
          wndclass.lpszMenuName   = NULL ;
          wndclass.lpszClassName  = szAppName ;

          if (!RegisterClass (&wndclass))
          {
               MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                           szAppName, MB_ICONERROR) ;
               return 0 ;
          }

          wndclass.lpfnWndProc    = ChildWndProc ;
          wndclass.cbWndExtra     = sizeof (long) ;
          wndclass.hIcon          = NULL ;
          wndclass.lpszClassName  = szChildClass ;

          RegisterClass (&wndclass) ;

          hwnd = CreateWindow (szAppName, TEXT ("Checker4 Mouse Hit-Test Demo"),
                               WS_OVERLAPPEDWINDOW,
                               CW_USEDEFAULT, CW_USEDEFAULT,
                               CW_USEDEFAULT, CW_USEDEFAULT,
                               NULL, NULL, hInstance, NULL) ;
          ShowWindow (hwnd, iCmdShow) ;
          UpdateWindow (hwnd) ;

          while (GetMessage (&msg, NULL, 0, 0))
          {
               TranslateMessage (&msg) ;
               DispatchMessage (&msg) ;
          }
          return msg.wParam ;
     }

     LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
     {
          static HWND hwndChild[DIVISIONS][DIVISIONS] ;
          int         cxBlock, cyBlock, x, y ;

          switch (message)
          {
          case WM_CREATE :
               for (x = 0 ; x < DIVISIONS ; x++)
```

```
          for (y = 0 ; y < DIVISIONS ; y++)
               hwndChild[x][y] = CreateWindow (szChildClass, NULL,
                         WS_CHILDWINDOW | WS_VISIBLE,
                         0, 0, 0, 0,
                         hwnd, (HMENU) (y << 8 | x),
                         (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE),
                         NULL) ;
     return 0 ;

case WM_SIZE :
     cxBlock = LOWORD (lParam) / DIVISIONS ;
     cyBlock = HIWORD (lParam) / DIVISIONS ;

     for (x = 0 ; x < DIVISIONS ; x++)
          for (y = 0 ; y < DIVISIONS ; y++)
               MoveWindow (hwndChild[x][y],
                           x * cxBlock, y * cyBlock,
                           cxBlock, cyBlock, TRUE) ;
     return 0 ;

case WM_LBUTTONDOWN :
     MessageBeep (0) ;
     return 0 ;

     // On set-focus message, set focus to child window

case WM_SETFOCUS:
     SetFocus (GetDlgItem (hwnd, idFocus)) ;
     return 0 ;

     // On key-down message, possibly change the focus window

case WM_KEYDOWN:
     x = idFocus & 0xFF ;
     y = idFocus >> 8 ;

     switch (wParam)
     {
     case VK_UP:      y-- ;                    break ;
     case VK_DOWN:    y++ ;                    break ;
     case VK_LEFT:    x-- ;                    break ;
     case VK_RIGHT:   x++ ;                    break ;
     case VK_HOME:    x = y = 0 ;              break ;
     case VK_END:     x = y = DIVISIONS - 1 ;  break ;
     default:         return 0 ;
     }
```

```
          x = (x + DIVISIONS) % DIVISIONS ;
          y = (y + DIVISIONS) % DIVISIONS ;

          idFocus = y << 8 | x ;

          SetFocus (GetDlgItem (hwnd, idFocus)) ;
          return 0 ;

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}

LRESULT CALLBACK ChildWndProc (HWND hwnd, UINT message,
                               WPARAM wParam, LPARAM lParam)
{
     HDC         hdc ;
     PAINTSTRUCT ps ;
     RECT        rect ;

     switch (message)
     {
     case WM_CREATE :
          SetWindowLong (hwnd, 0, 0) ;       // on/off flag
          return 0 ;

     case WM_KEYDOWN:
               // Send most key presses to the parent window

          if (wParam != VK_RETURN && wParam != VK_SPACE)
          {
               SendMessage (GetParent (hwnd), message, wParam, lParam) ;
               return 0 ;
          }
               // For Return and Space, fall through to toggle the square

     case WM_LBUTTONDOWN :
          SetWindowLong (hwnd, 0, 1 ^ GetWindowLong (hwnd, 0)) ;
          SetFocus (hwnd) ;
          InvalidateRect (hwnd, NULL, FALSE) ;
          return 0 ;

               // For focus messages, invalidate the window for repaint

     case WM_SETFOCUS:
          idFocus = GetWindowLong (hwnd, GWL_ID) ;
```

```
            // Fall through

    case WM_KILLFOCUS:
        InvalidateRect (hwnd, NULL, TRUE) ;
        return 0 ;

    case WM_PAINT :
        hdc = BeginPaint (hwnd, &ps) ;

        GetClientRect (hwnd, &rect) ;
        Rectangle (hdc, 0, 0, rect.right, rect.bottom) ;

            // Draw the "x" mark

        if (GetWindowLong (hwnd, 0))
        {
            MoveToEx (hdc, 0,          0, NULL) ;
            LineTo   (hdc, rect.right, rect.bottom) ;
            MoveToEx (hdc, 0,          rect.bottom, NULL) ;
            LineTo   (hdc, rect.right, 0) ;
        }
            // Draw the "focus" rectangle

        if (hwnd == GetFocus ())
        {
            rect.left   += rect.right / 10 ;
            rect.right  -= rect.left ;
            rect.top    += rect.bottom / 10 ;
            rect.bottom -= rect.top ;

            SelectObject (hdc, GetStockObject (NULL_BRUSH)) ;
            SelectObject (hdc, CreatePen (PS_DASH, 0, 0)) ;
            Rectangle (hdc, rect.left, rect.top, rect.right, rect.bottom) ;
            DeleteObject (SelectObject (hdc, GetStockObject (BLACK_PEN))) ;
        }

        EndPaint (hwnd, &ps) ;
        return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

您应该回忆得起来每一个子窗口有唯一的子窗口 ID 号,该 ID 号在 CreateWindow 调用创建窗口时定义。在 CHECKER3 中,此 ID 号是矩形的 x 和 y 位置的组合。一个程序可以通过下面的调用来获得一个特定子窗口的子窗口 ID:

```
idChild = GetWindowLong (hwndChild, GWL_ID) ;
```

下面的函数也有同样的功能：

```
idChild = GetDlgCtrlID (hwndChild) ;
```

正如函数名称所表示的，它主要用于对话框和控制窗口。如果您知道父窗口的句柄和子窗口 ID，此函数也可以获得子窗口的句柄：

```
hwndChild = GetDlgItem (hwndParent, idChild) ;
```

在 CHECKER4 中，全局变量 idFocus 用于保存有当前输入焦点窗口的子 ID 数。我在前面说过，当您在子窗口上面单击鼠标时，它们不会自动获得输入焦点。因此，CHECKER4 中的父窗口将通过调用下面的函数来处理 WM_SETFOCUS 消息：

```
SetFocus (GetDlgItem (hwnd, idFocus)) ;
```

这样设置输入一个子窗口的输入焦点。

ChildWndProc 处理 WM_SETFOCUS 和 WM_KILLFOCUS 消息。对于 WM_SETFOCUS，它将保存在全局变量 idFocus 中接收输入焦点的子窗口 ID。对于这两种消息，窗口是无效的，并产生一个 WM_PAINT 消息。如果 WM_PAINT 消息画出了有输入焦点的子窗口，则它将用 PS_DASH 笔的风格画一个矩形以表明此窗口有输入焦点。

ChildWndProc 也处理 WM_KEYDOWN 消息。对于除 Spacebar 和 Entert 键以外的其他消息，WM_KEYDOWN 都将给父窗口发送消息。另外，窗口过程也处理类似 WM_LBUTTONDOWN 消息的消息。

处理方向移动键是父窗口的代表。在风格相似的 CHECKER2 中，此程序可获得有输入焦点的子窗口的 x 和 y 坐标，并根据按下的特定的方向键来改变它们。然后通过调用 SetFocus 将输入焦点设置给新的子窗口。

7.5　捕获鼠标

一个窗口过程通常只在鼠标光标位于窗口的客户区或非客户区上时才接收鼠标消息。一个程序也可能需要当鼠标在窗口外时接收鼠标消息。如果是这样，程序就能“捕获”鼠标。别害怕，没事。

7.5.1　设计矩形

为了说明捕获鼠标的必要性，请让我们看一下 BLOKOUT1 程序（如程序 7-6 所示）。此程序看起来有一定的功能，但它却有十分严重的缺陷。

程序 7-6 BLOKOUT1 程序

BLOKOUT1.C

```
/*----------------------------------------------
   BLOKOUT1.C——Mouse Button Demo Program
              (c) Charles Petzold, 1998
  ----------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("BlokOut1") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Mouse Button Demo"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
```

```
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

void DrawBoxOutline (HWND hwnd, POINT ptBeg, POINT ptEnd)
{
     HDC hdc ;

     hdc = GetDC (hwnd) ;

     SetROP2 (hdc, R2_NOT) ;
     SelectObject (hdc, GetStockObject (NULL_BRUSH)) ;
     Rectangle (hdc, ptBeg.x, ptBeg.y, ptEnd.x, ptEnd.y) ;

     ReleaseDC (hwnd, hdc) ;
}
LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static BOOL  fBlocking, fValidBox ;
     static POINT ptBeg, ptEnd, ptBoxBeg, ptBoxEnd ;
     HDC          hdc ;
     PAINTSTRUCT ps ;

     switch (message)
     {
     case WM_LBUTTONDOWN :
          ptBeg.x = ptEnd.x = LOWORD (lParam) ;
          ptBeg.y = ptEnd.y = HIWORD (lParam) ;

          DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

          SetCursor (LoadCursor (NULL, IDC_CROSS)) ;

          fBlocking = TRUE ;
          return 0 ;

     case WM_MOUSEMOVE :
          if (fBlocking)
          {
               SetCursor (LoadCursor (NULL, IDC_CROSS)) ;

               DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
```

```
               ptEnd.x = LOWORD (lParam) ;
               ptEnd.y = HIWORD (lParam) ;

               DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
          }
          return 0 ;

     case WM_LBUTTONUP :
          if (fBlocking)
          {
               DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

               ptBoxBeg   = ptBeg ;
               ptBoxEnd.x = LOWORD (lParam) ;
               ptBoxEnd.y = HIWORD (lParam) ;

               SetCursor (LoadCursor (NULL, IDC_ARROW)) ;
               fBlocking = FALSE ;
               fValidBox = TRUE ;

               InvalidateRect (hwnd, NULL, TRUE) ;
          }
          return 0 ;

     case WM_CHAR :
          if (fBlocking & wParam == '\x1B')       // i.e., Escape
          {
               DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

               SetCursor (LoadCursor (NULL, IDC_ARROW)) ;

               fBlocking = FALSE ;
          }
          return 0 ;

     case WM_PAINT :
          hdc = BeginPaint (hwnd, &ps) ;

          if (fValidBox)
          {
               SelectObject (hdc, GetStockObject (BLACK_BRUSH)) ;
               Rectangle (hdc, ptBoxBeg.x, ptBoxBeg.y,
                    ptBoxEnd.x, ptBoxEnd.y) ;
          }

          if (fBlocking)
          {
```

```
            SetROP2 (hdc, R2_NOT) ;
            SelectObject (hdc, GetStockObject (NULL_BRUSH)) ;
            Rectangle (hdc, ptBeg.x, ptBeg.y, ptEnd.x, ptEnd.y) ;
        }

        EndPaint (hwnd, &ps) ;
        return 0 ;

    case WM_DESTROY :
        PostQuitMessage (0) ;
        return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

此程序表明了一点，它可以在 Windows 的“画图”程序中执行。由按下鼠标左键开始确定矩形的一角，然后拖动鼠标。程序将画一个矩形的轮廓，其相对位置是鼠标当前的位置。当您释放鼠标后，程序将填充这个矩形。图 7-4 显示了一个已经画完的矩形和另一个正在画的矩形。

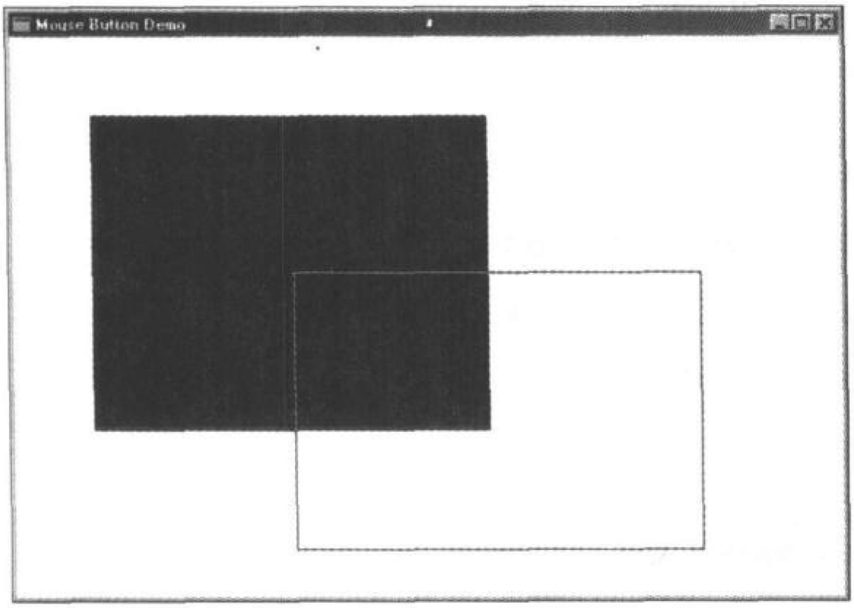


图 7-4　BLOKOUT1 的屏幕显示

这样，问题在哪里呢?

请试一试下边的操作：在 BLOKOUT1 的客户区按下鼠标的左键，然后将光标移出窗口。程序将停止接收 WM_MOUSEMOVE 消息。现在释放键，BLOKOUT1 将不再获得 WM_BUTTONUP 消息，因为光标在客户区以外。将光标移回 BLOKOUT1 的客户区。窗

口过程仍然认为键处于按下状态。

这样不好。程序不知道发生了什么。

7.5.2 捕获的解决方案

BLOKOUT1 显示了一些常见的程序功能，但它的代码显然有缺陷。这就是要引入鼠标捕获来对付的一类问题。如果用户正在拖动鼠标，那么当鼠标短时间内被拖出窗口时应该没有什么大问题，程序应该仍然控制着鼠标。

捕获鼠标要比放置一个老鼠夹子容易一些，您只要调用

```
SetCapture (hwnd) ;
```

在这个函数调用之后，Windows 将所有鼠标消息发给窗口句柄为 hwnd 的窗口过程。鼠标消息总是客户区消息，即使鼠标正在窗口的非客户区。lParam 参数将指示鼠标在客户区坐标中的位置。不过，当鼠标位于客户区的左边或者上方时，这些 x 和 y 坐标可以是负的。当您想释放鼠标时，调用

```
ReleaseCapture () ;
```

从而使处理恢复正常。

在 32 位的 Windows 中，鼠标捕获要比在以前的 Windows 版本中限制多一些。特别地，如果鼠标被捕获，而鼠标键目前并未被按下，并且鼠标光标移到了另一个窗口上，那么将不是由捕获鼠标的那个窗口而是由光标下面的窗口来接收鼠标消息。对于防止一个程序在捕获鼠标之后不释放它而引起整个系统的混乱，这是必要的。

换句话说，只有当鼠标键在您的客户区中被按下时才捕获鼠标；当键被释放时，才释放鼠标捕获。

7.5.3 BLOKOUT2 程序

演示鼠标捕获的 BLOKOUT2 程序如程序 7-7 所示。

程序 7-7 BLOKOUT2 程序

BLOKOUT2.C

```
/*-------------------------------------------------------
    BLOKOUT2.C——Mouse Button & Capture Demo Program
                (c) Charles Petzold, 1998
  -------------------------------------------------------*/

#include <windows.h>
```

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("BlokOut2") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Mouse Button & Capture Demo"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

void DrawBoxOutline (HWND hwnd, POINT ptBeg, POINT ptEnd)
{
```

```
     HDC hdc ;

     hdc = GetDC (hwnd) ;

     SetROP2 (hdc, R2_NOT) ;
     SelectObject (hdc, GetStockObject (NULL_BRUSH)) ;
     Rectangle (hdc, ptBeg.x, ptBeg.y, ptEnd.x, ptEnd.y) ;

     ReleaseDC (hwnd, hdc) ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static BOOL  fBlocking, fValidBox ;
     static POINT ptBeg, ptEnd, ptBoxBeg, ptBoxEnd ;
     HDC          hdc ;
     PAINTSTRUCT ps ;

     switch (message)
     {
     case WM_LBUTTONDOWN :
          ptBeg.x = ptEnd.x = LOWORD (lParam) ;
          ptBeg.y = ptEnd.y = HIWORD (lParam) ;

          DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

          SetCapture (hwnd) ;
          SetCursor (LoadCursor (NULL, IDC_CROSS)) ;

          fBlocking = TRUE ;
          return 0 ;

     case WM_MOUSEMOVE :
          if (fBlocking)
          {
               SetCursor (LoadCursor (NULL, IDC_CROSS)) ;

               DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

               ptEnd.x = LOWORD (lParam) ;
               ptEnd.y = HIWORD (lParam) ;

               DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
          }
          return 0 ;

     case WM_LBUTTONUP :
          if (fBlocking)
```

```
          {
               DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

               ptBoxBeg   = ptBeg ;
               ptBoxEnd.x = LOWORD (lParam) ;
               ptBoxEnd.y = HIWORD (lParam) ;

               ReleaseCapture () ;
               SetCursor (LoadCursor (NULL, IDC_ARROW)) ;

               fBlocking = FALSE ;
               fValidBox = TRUE ;

               InvalidateRect (hwnd, NULL, TRUE) ;
          }
          return 0 ;

     case WM_CHAR :
          if (fBlocking & wParam == '\x1B')       // i.e., Escape
          {
               DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
               ReleaseCapture () ;
               SetCursor (LoadCursor (NULL, IDC_ARROW)) ;

               fBlocking = FALSE ;
          }
          return 0 ;

     case WM_PAINT :
          hdc = BeginPaint (hwnd, &ps) ;

          if (fValidBox)
          {
               SelectObject (hdc, GetStockObject (BLACK_BRUSH)) ;
               Rectangle (hdc, ptBoxBeg.x, ptBoxBeg.y,
                    ptBoxEnd.x, ptBoxEnd.y) ;
          }

          if (fBlocking)
          {
               SetROP2 (hdc, R2_NOT) ;
               SelectObject (hdc, GetStockObject (NULL_BRUSH)) ;
               Rectangle (hdc, ptBeg.x, ptBeg.y, ptEnd.x, ptEnd.y) ;
          }

          EndPaint (hwnd, &ps) ;
          return 0 ;
```

```
        case WM_DESTROY :
            PostQuitMessage (0) ;
            return 0 ;
        }
        return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

BLOKOUT2 程序和 BLOKOUT1 程序一样，只是多了三行新代码：在 WM_LBUTTONDOWN 消息期间调用 SetCapture，在 WM_LBUTTONDOWN 和 WM_CHAR 消息期间调用 ReleaseCapture。检查画出窗口：使窗口小于屏幕大小，开始在客户区画出一块矩形，然后将鼠标光标移出客户区的右边或下边，最后释放鼠标键。程序将获得整个矩形的坐标。只是需要扩大窗口才能看清它。

捕获鼠标并非只适用于那些古怪的应用程序。如果您需要每当在客户区按下一个鼠标键时都能够跟踪 WM_MOUSEMOVE 消息，并直到鼠标键被释放为止，那么您就应该捕获鼠标。这样将简化您的程序，同时又符合用户的期望。

7.6 鼠标轮

与传统的鼠标相比，Microsoft IntelliMouse 的特点是在两个键之间多了一个小轮。您可以按下这个轮，这时它的功能相当于鼠标键的中键；或者您也可以用食指来转动它，这会产生一个特殊的消息，叫做 WM_MOUSEWHEEL。使用鼠标轮的程序通过滚动或放大文档来响应此消息。它最初听起来像一个不必要的暗机关，但我必须承认，我很快就习惯于使用鼠标轮来滚动 Microsoft Word 和 Microsoft Internet Explorer 了。

我不想讨论鼠标轮的全部使用方法。实际上，我只是想告诉您如何在现有的程序（例如程序 SYSMETS4）中添加鼠标轮逻辑，以便在客户区中滚动数据。最终的 SYSMETS 程序如程序 7-8 所示。

程序 7-8 SYSMETS 程序

SYSMETS.C

```
/*-------------------------------------------------------
    SYSMETS.C——Final System Metrics Display Program
                (c) Charles Petzold, 1998
  -------------------------------------------------------*/

#include <windows.h>
#include "sysmets.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
```

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("SysMets") ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASS    wndclass ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Get System Metrics"),
                          WS_OVERLAPPEDWINDOW | WS_VSCROLL | WS_HSCROLL,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static int  cxChar, cxCaps, cyChar, cxClient, cyClient, iMaxWidth ;
     static int  iDeltaPerLine, iAccumDelta ;     // for mouse wheel logic
     HDC         hdc ;
```

```
int            i, x, y, iVertPos, iHorzPos, iPaintBeg, iPaintEnd ;
PAINTSTRUCT    ps ;
SCROLLINFO     si ;
TCHAR          szBuffer[10] ;
TEXTMETRIC     tm ;
ULONG          ulScrollLines ;                  // for mouse wheel logic
switch (message)
{
case WM_CREATE:
    hdc = GetDC (hwnd) ;

    GetTextMetrics (hdc, &tm) ;
    cxChar = tm.tmAveCharWidth ;
    cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
    cyChar = tm.tmHeight + tm.tmExternalLeading ;

    ReleaseDC (hwnd, hdc) ;

        // Save the width of the three columns

    iMaxWidth = 40 * cxChar + 22 * cxCaps ;

        // Fall through for mouse wheel information

case WM_SETTINGCHANGE:
    SystemParametersInfo (SPI_GETWHEELSCROLLLINES, 0, &ulScrollLines, 0) ;

        // ulScrollLines usually equals 3 or 0 (for no scrolling)
        // WHEEL_DELTA equals 120, so iDeltaPerLine will be 40

    if (ulScrollLines)
         iDeltaPerLine = WHEEL_DELTA / ulScrollLines ;
    else
         iDeltaPerLine = 0 ;

    return 0 ;

case WM_SIZE:
    cxClient = LOWORD (lParam) ;
    cyClient = HIWORD (lParam) ;

        // Set vertical scroll bar range and page size

    si.cbSize = sizeof (si) ;
    si.fMask  = SIF_RANGE | SIF_PAGE ;
    si.nMin   = 0 ;
    si.nMax   = NUMLINES - 1 ;
    si.nPage  = cyClient / cyChar ;
```

```
          SetScrollInfo (hwnd, SB_VERT, &si, TRUE) ;

               // Set horizontal scroll bar range and page size

          si.cbSize = sizeof (si) ;
          si.fMask  = SIF_RANGE | SIF_PAGE ;
          si.nMin   = 0 ;
          si.nMax   = 2 + iMaxWidth / cxChar ;
          si.nPage  = cxClient / cxChar ;
          SetScrollInfo (hwnd, SB_HORZ, &si, TRUE) ;
          return 0 ;

     case WM_VSCROLL:
               // Get all the vertical scroll bar information

          si.cbSize = sizeof (si) ;
          si.fMask  = SIF_ALL ;
          GetScrollInfo (hwnd, SB_VERT, &si) ;

               // Save the position for comparison later on

          iVertPos = si.nPos ;

          switch (LOWORD (wParam))
          {
          case SB_TOP:
               si.nPos = si.nMin ;
               break ;

          case SB_BOTTOM:
               si.nPos = si.nMax ;
               break ;

          case SB_LINEUP:
               si.nPos -= 1 ;
               break ;

          case SB_LINEDOWN:
               si.nPos += 1 ;
               break ;

          case SB_PAGEUP:
               si.nPos -= si.nPage ;
               break ;

          case SB_PAGEDOWN:
               si.nPos += si.nPage ;
```

```
                break ;

            case SB_THUMBTRACK:
                si.nPos = si.nTrackPos ;
                break ;
            default:
                break ;
            }
                // Set the position and then retrieve it. Due to adjustments
                //  by Windows it may not be the same as the value set.

            si.fMask = SIF_POS ;
            SetScrollInfo (hwnd, SB_VERT, &si, TRUE) ;
            GetScrollInfo (hwnd, SB_VERT, &si) ;

                // If the position has changed, scroll the window and update it
            if (si.nPos != iVertPos)
            {
                 ScrollWindow (hwnd, 0, cyChar * (iVertPos - si.nPos),
                                     NULL, NULL) ;
                 UpdateWindow (hwnd) ;
            }
            return 0 ;

        case WM_HSCROLL:
                // Get all the vertical scroll bar information

            si.cbSize = sizeof (si) ;
            si.fMask  = SIF_ALL ;

                // Save the position for comparison later on

            GetScrollInfo (hwnd, SB_HORZ, &si) ;
            iHorzPos = si.nPos ;

            switch (LOWORD (wParam))
            {
            case SB_LINELEFT:
                si.nPos -= 1 ;
                break ;

            case SB_LINERIGHT:
                si.nPos += 1 ;
                break ;

            case SB_PAGELEFT:
                si.nPos -= si.nPage ;
```

```
                break ;

            case SB_PAGERIGHT:
                si.nPos += si.nPage ;
                break ;

            case SB_THUMBPOSITION:
                si.nPos = si.nTrackPos ;
                break ;

            default:
                break ;
            }
                // Set the position and then retrieve it. Due to adjustments
                //  by Windows it may not be the same as the value set.

            si.fMask = SIF_POS ;
            SetScrollInfo (hwnd, SB_HORZ, &si, TRUE) ;
            GetScrollInfo (hwnd, SB_HORZ, &si) ;

                // If the position has changed, scroll the window

            if (si.nPos != iHorzPos)
            {
                 ScrollWindow (hwnd, cxChar * (iHorzPos - si.nPos), 0,
                               NULL, NULL) ;
            }
            return 0 ;

        case WM_KEYDOWN :
            switch (wParam)
            {
            case VK_HOME :
                SendMessage (hwnd, WM_VSCROLL, SB_TOP, 0) ;
                break ;

            case VK_END :
                SendMessage (hwnd, WM_VSCROLL, SB_BOTTOM, 0) ;
                break ;

            case VK_PRIOR :
                SendMessage (hwnd, WM_VSCROLL, SB_PAGEUP, 0) ;
                break ;

            case VK_NEXT :
                SendMessage (hwnd, WM_VSCROLL, SB_PAGEDOWN, 0) ;
                break ;
```

```
          case VK_UP :
               SendMessage (hwnd, WM_VSCROLL, SB_LINEUP, 0) ;
               break ;

          case VK_DOWN :
               SendMessage (hwnd, WM_VSCROLL, SB_LINEDOWN, 0) ;
               break ;

          case VK_LEFT :
               SendMessage (hwnd, WM_HSCROLL, SB_PAGEUP, 0) ;
               break ;

          case VK_RIGHT :
               SendMessage (hwnd, WM_HSCROLL, SB_PAGEDOWN, 0) ;
               break ;
          }
          return 0 ;

     case WM_MOUSEWHEEL:
          if (iDeltaPerLine == 0)
             break ;
          iAccumDelta += (short) HIWORD (wParam) ;     // 120 or -120

          while (iAccumDelta >= iDeltaPerLine)
          {
               SendMessage (hwnd, WM_VSCROLL, SB_LINEUP, 0) ;
               iAccumDelta -= iDeltaPerLine ;
          }

          while (iAccumDelta <= -iDeltaPerLine)
          {
               SendMessage (hwnd, WM_VSCROLL, SB_LINEDOWN, 0) ;
               iAccumDelta += iDeltaPerLine ;
          }

          return 0 ;

     case WM_PAINT :
          hdc = BeginPaint (hwnd, &ps) ;

               // Get vertical scroll bar position

          si.cbSize = sizeof (si) ;
          si.fMask  = SIF_POS ;
          GetScrollInfo (hwnd, SB_VERT, &si) ;
          iVertPos = si.nPos ;
```

```
          // Get horizontal scroll bar position

     GetScrollInfo (hwnd, SB_HORZ, &si) ;
     iHorzPos = si.nPos ;

          // Find painting limits

     iPaintBeg = max (0, iVertPos + ps.rcPaint.top / cyChar) ;
     iPaintEnd = min (NUMLINES - 1,
                      iVertPos + ps.rcPaint.bottom / cyChar) ;

     for (i = iPaintBeg ; i <= iPaintEnd ; i++)
     {
          x = cxChar * (1 - iHorzPos) ;
          y = cyChar * (i - iVertPos) ;

          TextOut (hdc, x, y,
                   sysmetrics[i].szLabel,
                   lstrlen (sysmetrics[i].szLabel)) ;

          TextOut (hdc, x + 22 * cxCaps, y,
                   sysmetrics[i].szDesc,
                   lstrlen (sysmetrics[i].szDesc)) ;

          SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;

          TextOut (hdc, x + 22 * cxCaps + 40 * cxChar, y, szBuffer,
                   wsprintf (szBuffer, TEXT ("%5d"),
                        GetSystemMetrics (sysmetrics[i].iIndex))) ;

          SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
     }

     EndPaint (hwnd, &ps) ;
     return 0 ;

case WM_DESTROY :
     PostQuitMessage (0) ;
     return 0 ;
}
return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

转动轮会导致 Windows 在有输入焦点的窗口(不是鼠标光标下面的窗口)产生 WM_MOUSEWHEEL 消息。与平常一样,lParam 将获得鼠标的位置,当然坐标是相对于屏幕左上角的,而不是客户区的。另外,wParam 的低字包含一系列的标识,用于表明鼠标键和 Shift 与 Ctrl 键的状态。

新的信息保存在 wParam 的高字。其中有一个“delta”值,该值当前可以是 120 或 -120,这取决于轮的向前转动(也就是说,向鼠标的前面,即带有按钮与电缆的一端)还是向后转动。值 120 或 -120 表明文档将分别向上或向下滚动三行。这里的思想是,以后版本的鼠标轮能有比现在的鼠标更好的等级,并且用 delta 值,例如 40 和 -40,来产生 WM_MOUSEWHEEL 消息。这些值能使文档只向上或向下滚动一行。

为使程序只适用于常规环境,SYSMETS 将调用 SystemParametersInfo,并在 WM_CREATE 和 WM_SETTINGCHANGE 消息时带有 SPI_GETWHEELSCROLLLINES。此值说明 WHEEL_DELTA 的 delta 值将滚动多少行,WHEEL_DELTA 在 WINUSER.H 中定义。WHEEL_DELTA 等于 120,并且默认情况下 SystemParametersInfo 返回 3,因此与滚动一行相联系的 delta 值就是 40。SYSMETS 将此值保存在 iDeltaPerLine。

在 WM_MOUSEWHEEL 消息期间,SYSMETS 将 delta 值加到静态变量 iAccumDelta 上。然后,如果 iAccumDelta 大于或等于 iDeltaPerLine(或者是小于或等于 - iDeltaPerLin),SYSMETS 用 SB_LINEUP 或 SB_LINEDOWN 值产生 WM_VSCROLL 消息。对于每一个 WM_VSCROLL 消息,iAccumDelta 由 iDeltaPerLine 增加(或减少)。此代码允许 delta 值大于、小于或等于滚动一行所需要的 delta 值。

7.6.1 后会有期

只有一个非标准的鼠标问题:创建自定义鼠标光标。我将在第十章,与其他 Windows 资源一起讨论此问题。

第八章

计时器

Microsoft Windows 计时器是一种输入设备,它周期性地每经过一个指定的时间间隔就通知应用程序一次。您的程序将时间间隔告诉 Windows,例如"每 10 秒钟通知我一声",然后 Windows 给您的程序发送周期性发生的 WM_TIMER 消息以表示时间到了。

初看起来,Windows 计时器似乎不如键盘和鼠标设备重要,而且对许多应用程序来说确实如此。但是,计时器比您可能认为的要重要得多,它不只用于监视程序,比如出现在工具栏中的 Windows 时钟和这一章包括的两个时钟程序。下面是 Windows 计时器的其他应用,有些可能并不那么明显:

- **多任务** 虽然 Windows 98 是一个抢占式的多任务环境,有时程序尽快将控制返回给 Windows 会效率更高。如果一个程序必须进行大量的处理,那么它可以将作业分成小块,每接收到一条 WM_TIMER 消息处理一块(我将在第二十章中对此做更多的讨论)。
- **维护更新过的状态报告** 程序可以利用计时器来显示持续变化信息的"实时"更新,比如关于系统资源的变化或某个任务的进展情况。
- **实现"自动保存"功能** 计时器提示 Windows 程序在指定的时间过去后把用户的工作保存在磁盘上。
- **终止程序演示版本的运行** 一些程序的演示版本被设计成在其开始后多长时间,比如说,30 分钟结束。如果时间已到,那么计时器就会通知应用程序。
- **步进移动** 游戏中的图形对象或计算机辅助教学程序中的连续显示需要按指定的速率来处理。利用计时器可以消除由于微处理器速度不同而造成的不一致。
- **多媒体** 播放 CD 音频、声音或音乐的程序通常在后台播放音频数据。一个程序可以使用计时器来周期性地检查已播放了多少音频数据,并据此协调屏幕上的视觉信息。

计时器的另一个应用是,可以保证程序在退出窗口过程后能够重新得到控制。在通常情况下,程序不能够知道何时下一个消息到来。

8.1 计时器基础

您可以通过调用 SetTimer 函数为您的 Windows 程序分配一个计时器。SetTimer 有一个时间间隔范围为 1～4 294 967 295 毫秒(将近 50 天)的整型参数,这个值指示 Windows 每隔多长时间给您的程序发送 WM_TIMER 消息。例如,如果间隔为 1000 毫秒,那么 Windows 将每秒给程序发送一个 WM_TIMER 消息。

当您的程序用完计时器时,它调用 KillTimer 函数停止计时器消息。在处理 WM_TIMER 消息时,您可以通过调用 KillTimer 函数来编写一个"瞬间"计时器。KillTimer 调用清除消息队列中尚未被处理的 WM_TIMER 消息,从而使程序在调用 KillTimer 之后就不会再接收到 WM_TIMER 消息。

8.1.1 系统和计时器

Windows 计时器是 PC 的硬件和 ROM BIOS 构造的计时器逻辑的一种相对简单的扩展。回到 Windows 前的 MS-DOS 编程,应用程序能够通过捕获称为"计时器滴答"的 BIOS 中断来实现时钟或计时器。这些中断每 54.915 毫秒产生一次,或者大约每秒 18.2 次。一些为 MS-DOS 编写的程序自己捕获这个硬件中断以实现时钟和计时器。这是原始的 IBM PC 的微处理器时钟主频值 4.772720 MHz 被 2^{18} 所除而得出的结果。

Windows 应用程序不捕获 BIOS 中断,相反,Windows 本身处理硬件中断,这样应用程序就不必进行处理。对于目前拥有计时器的每个程序,Windows 保存一个每次硬件计时器滴答减少的计数。当这个计数减到 0 时,Windows 在适当的应用程序消息队列中放置一个 WM_TIMER 消息,并将计数重新置为其最初的值。

因为 Windows 应用程序从正常的消息队列中获取 WM_TIMER 消息,所以您的程序在进行其他处理时不必担心被偶然的 WM_TIMER 消息中断了程序。在这方面,计时器类似于键盘和鼠标。驱动程序处理异步硬件中断事件,Windows 把这些事件翻译为规整的、结构化和顺序化的消息。

在 Windows 98 中,计时器与潜在的 PC 计时器一样具有 55 毫秒的分辨率。在 Microsoft Windows NT 中,计时器的分辨率为 10 毫秒。

Windows 应用程序不能以高于这一分辨率的速率(在 Windows 98 下每秒 18.2 次,在 Windows NT 下每秒大约 100 次)接收 WM_TIMER 消息。在 SetTimer 调用中指定的时间间隔总是截尾为时钟滴答的整数倍。例如,1000 毫秒的间隔除以 54.925 毫秒,得到 18.207 个时钟滴答,截尾后为 18 个时钟滴答,它实际上为 989 毫秒。对每个小于 55 毫秒

的间隔，每个时钟滴答都产生一个 WM_TIMER 消息。

8.1.2 计时器消息不是异步的

因为计时器是基于硬件计时器中断，程序员有时会误入歧途，认为他们的程序会异步地被中断来处理 WM_TIMER 消息。

然而，WM_TIMER 消息却不是异步的。WM_TIMER 消息放在正常的消息队列之中，和其他消息一起参加排序，因此，如果在 SetTimer 调用中指定间隔为 1000 毫秒，那么不能保证程序每 1000 毫秒或者 989 毫秒就会收到一个 WM_TIMER 消息。如果其他程序的运行事件超过一秒，在此期间，您的程序将收不到任何 WM_TIMER 消息。您可以使用本章的程序来演示这一点。事实上，Windows 对 WM_TIMER 消息的处理非常类似于对 WM_PAINT 消息的处理，这两个消息都是低优先级的，程序只有在消息队列中没有其他消息时才接收它们。

WM_TIMER 还在另一方面同 WM_PAINT 相似：Windows 不能持续向消息队列中放入多个 WM_TIMER 消息，而是将多余的 WM_TIMER 消息组合成一个消息。因此，应用程序不会一次收到多个这样的消息，尽管可能在短时间内得到两个 WM_TIMER 消息。应用程序不能确定这种处理所导致的 WM_TIMER 消息"丢失"的数目。

这样，WM_TIMER 消息仅仅在需要更新时才提示程序，程序本身不能通过统计 WM_TIMER 消息的数目来保持时间（在本章后面，我们将编写两个每秒更新一次的时钟程序，并可以看到如何做到这一点）。

为方便起见，下面在讨论时钟时，我将使用"每秒得到一次 WM_TIMER 消息"这样的词句，但是请记住，这些消息并非精确的时钟滴答中断。

8.2 计时器的使用：三种方法

如果您需要在整个程序期间使用计时器，那么您将或者从 WinMain 函数中或者在处理 WM_CREATE 消息时调用 SetTimer，并在退出 WinMain 或响应 WM_DESTROY 消息时调用 KillTimer。根据调用 SetTimer 时使用的参数，可以用以下三种方法之一使用计时器。

8.2.1 方法一

这是最方便的一种方法，它让 Windows 把 WM_TIMER 消息发送到应用程序的正常窗口过程中。SetTimer 调用如下所示：

```
SetTimer (hwnd, 1, uiMsecInterval, NULL) ;
```

第一个参数是其窗口过程将接收 WM_TIMER 消息的窗口的句柄。第二个参数是计时器 ID,它是一个非 0 数,在整个例子中假定为 1。第三个参数是一个 32 位无符号整数,以毫秒为单位指定一个时间间隔,一个 60 000 的值将使 Windows 每分钟发送一次 WM_TIMER 消息。

您可以通过调用

```
KillTimer (hwnd, 1) ;
```

在任何时刻停止 WM_TIMER 消息(即使正在处理 WM_TIMER 消息)。此函数的第二个参数是 SetTimer 调用中所用的同一个计时器 ID。在终止程序之前,您应该响应 WM_DESTROY 消息以停止任何活动的计时器。

当您的窗口过程收到一个 WM_TIMER 消息时,wParam 参数等于计时器的 ID 值(上述情形为 1),lParam 参数为 0。如果需要设置多个计时器,那么对每个计时器都使用不同的计时器 ID。wParam 的值将随传递到窗口过程的 WM_TIMER 消息的不同而不同。为了使程序具有更好的可读性,您可以使用 # define 语句定义不同的计时器 ID:

```
# define TIMER_SEC 1
# define TIMER_MIN 2
```

然后您可以使用两个 SetTimer 调用来设置两个计时器:

```
SetTimer (hwnd, TIMER_SEC, 1000, NULL) ;
SetTimer (hwnd, TIMER_MIN, 60000, NULL) ;
```

WM_TIMER 的处理如下所示:

```
case WM_TIMER:
     switch (wParam)
     {
     case TIMER_SEC:
          [once-per-second processing]
          break ;
     case TIMER_MIN:
          [once-per-second processing]
          break ;
     }
     return 0 ;
```

如果您要将一个已经存在的计时器设置为不同的时间间隔,您可以简单地用不同的时间值再次调用 SetTimer。在时钟程序里,如果显示秒或不显示秒是可选的,您就可以这样做,只需简单地将时间间隔在 1000 毫秒和 60 000 毫秒间切换就可以了。

程序 8-1 显示了一个使用计时器的简单程序,名为 BEEPER1,计时器的时间间隔设

置为 1 秒。当它收到 WM_TIMER 消息时,它将客户区的颜色由蓝色变到红色或由红色变到蓝色,并通过调用 MessageBeep 函数发出响声。(虽然 MessageBeep 通常用于 MessageBox,但它确实是一个全功能的鸣叫函数。在有声卡的 PC 机上,一般可以使用不同的 MB_ICON 参数作为 MessageBeep 的一个参数用于 MessageBox,来播放用户在"控制面板"的"声音"程序中选择的不同声音。)

BEEPER1 在窗口过程处理 WM_CREATE 消息时设置计时器。在处理 WM_TIMER 消息期间,BEEPER1 调用 MessageBeep,翻转 bFlipFlop 的值并使窗口无效以产生 WM_PAINT 消息。在处理 WM_PAINT 消息期间,BEEPER1 通过调用 GetClientRect 获得窗口大小的 RECT 结构,通过调用 FillRect 改变窗口的颜色(见程序 8-1)。

程序 8-1 BEEPER1 程序

BEEPER1.C

```
/*---------------------------------------
   BEEPER1.C——Timer Demo Program No. 1
              (c) Charles Petzold, 1998
  ---------------------------------------*/

#include <windows.h>

#define ID_TIMER 1

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("Beeper1") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
```

```
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Beeper1 Timer Demo"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static BOOL  fFlipFlop = FALSE ;
     HBRUSH       hBrush ;
     HDC          hdc ;
     PAINTSTRUCT ps ;
     RECT         rc ;

     switch (message)
     {
     case WM_CREATE:
          SetTimer (hwnd, ID_TIMER, 1000, NULL) ;
          return 0 ;

     case WM_TIMER :
          MessageBeep (-1) ;
          fFlipFlop = !fFlipFlop ;
          InvalidateRect (hwnd, NULL, FALSE) ;
          return 0 ;

     case WM_PAINT :
          hdc = BeginPaint (hwnd, &ps) ;

          GetClientRect (hwnd, &rc) ;
```

```
          hBrush = CreateSolidBrush (fFlipFlop ? RGB(255,0,0) : RGB(0,0,255)) ;
          FillRect (hdc, &rc, hBrush) ;

          EndPaint (hwnd, &ps) ;
          DeleteObject (hBrush) ;
          return 0 ;

     case WM_DESTROY :
          KillTimer (hwnd, ID_TIMER) ;
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

因为 BEEPER1 每次收到 WM_TIMER 消息时,都用颜色的变换指出来,所以您可以通过调用 BEEPER1 来查看 WM_TIMER 消息的性质,并完成 Windows 内部的一些其他操作。

例如,首先调用"控制面板"的"显示"程序,选择"效果",确定"拖动窗口时显示其内容"复选框没有被选中。现在,试着移动或者缩放 BEEPER1 窗口,这将导致程序进入"模态消息循环"。Windows 通过在内部消息而非您程序的消息循环中捕获所有消息来禁止对移动或者缩放操作的任何干扰。通过此循环到达程序窗口的大多数消息都被丢弃,这就是 BEEPER1 停止蜂鸣的原因。当完成了移动与缩放之后,您将会注意到 BEEPER1 不能取得它所丢弃的所有 WM_TIMER 消息,尽管前两个消息的间隔可能少于 1 秒。

在"拖动窗口时显示其内容"复选框被选中时,Windows 中的模态消息循环会试图给您的窗口过程传递一些丢失的消息。这样做有时工作得很好,有时却不行。

8.2.2 方法二

设置计时器的第一种方法是把 WM_TIMER 消息发送到通常的窗口过程,而第二种方法是让 Windows 直接将计时器消息发送给您程序的另一个函数。

接收这些计时器消息的函数被称为"回调"函数,这是一个在您的程序之中但是由 Windows 调用的函数。您先告诉 Windows 此函数的地址,然后 Windows 调用此函数。这看起来也很熟悉,因为程序的窗口过程实际上也是一类回调函数。当注册窗口类时,要将函数的地址告诉 Windows,当发送消息给程序时,Windows 调用此函数。

SetTimer 并非是唯一使用回调函数的 Windows 函数。CreateDialog 和 DialogBox 函数(将在第 11 章中介绍)使用回调函数处理对话框中的消息;有几个 Windows 函数(EnumChildWindow、EnumFonts、EnumObjects、EnumProps 和 EnumWindow)把枚举信息传递给回调

函数;还有几个不那么常用的函数(GrayString,LineDDA 和 SetWindowHookEx)也要求回调函数。

像窗口过程一样,回调函数也必须定义为 CALLBACK,因为它是由 Windows 从程序的代码段调用的。回调函数的参数和回调函数的返回值依赖于回调函数的目的。同计时器有关的回调函数中,输入参数同窗口过程的输入参数一样。计时器回调函数不向 Windows 返回值。

我们把以下的回调函数称为 TimerProc(您能够选择与其他一些用语不发生冲突的任何名称),它只处理 WM_TIMER 消息:

```
VOID CALLBACK TimerProc (HWND hwnd, UINT message, UINT iTimerID, DWORD dwTime)
{
    [process WM_TIMER message]
}
```

TimerProc 的参数 hwnd 是在调用 SetTimer 时指定的窗口句柄。Windows 只把 WM_TIMER 消息送给 TimerProc,因此消息参数总是等于 WM_TIMER。iTimerID 值是计时器 ID,dwTimer 值是与从 GetTickCount 函数的返回值兼容的值。这是自 Windows 启动后所经过的毫秒数。

在 BEEPER1 中已经看到过,用第一种方法设置计时器时要求下面格式的 SetTimer 调用:

```
SetTimer (hwnd, iTimerID, iMsecInterval, NULL) ;
```

您使用回调函数处理 WM_TIMER 消息时,SetTimer 的第 4 个参数由回调函数的地址取代,如下所示:

```
SetTimer (hwnd, iTimerID, iMsecInterval, TimerProc) ;
```

我们来看看一些示例代码,这样您就会了解这些东西是如何组合在一起的。在功能上,除了 Windows 发送一个计时器消息给 TimerProc 而非 WndProc 之外,程序 8-2 所示的 BEEPER2 程序与 BEEPER1 是相同的。注意,TimerProc 和 WndProc 一起被声明在程序的开始处。

程序 8-2 BEEPER2 程序

BEEPER2.C

```
/*---------------------------------------
    BEEPER2.C——Timer Demo Program No. 2
              (c) Charles Petzold, 1998
  ---------------------------------------*/
```

```
#include <windows.h>

#define ID_TIMER    1

LRESULT CALLBACK WndProc  (HWND, UINT, WPARAM, LPARAM) ;
VOID    CALLBACK TimerProc (HWND, UINT, UINT,   DWORD ) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static char szAppName[] = "Beeper2" ;
     HWND     hwnd ;
     MSG      msg ;
     WNDCLASSwndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, "Beeper2 Timer Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
```

```
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     switch (message)
     {
     case WM_CREATE:
          SetTimer (hwnd, ID_TIMER, 1000, TimerProc) ;
          return 0 ;

     case WM_DESTROY:
          KillTimer (hwnd, ID_TIMER) ;
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}

VOID CALLBACK TimerProc (HWND hwnd, UINT message, UINT iTimerID, DWORD dwTime)
{
     static BOOL fFlipFlop = FALSE ;
     HBRUSH      hBrush ;
     HDC         hdc ;
     RECT        rc ;

     MessageBeep (-1) ;
     fFlipFlop = !fFlipFlop ;

     GetClientRect (hwnd, &rc) ;

     hdc = GetDC (hwnd) ;
     hBrush = CreateSolidBrush (fFlipFlop ? RGB(255,0,0) : RGB(0,0,255)) ;

     FillRect (hdc, &rc, hBrush) ;
     ReleaseDC (hwnd, hdc) ;
     DeleteObject (hBrush) ;
}
```

8.2.3 方法三

设置计时器的第三种方法类似于第二种方法，只是传递给 SetTimer 的 hwnd 参数被设置为 NULL，并且第二个参数(通常为计时器 ID)被忽略了，最后，此函数返回计时器 ID：

```
iTimerID = SetTimer (NULL, 0, wMsecInterval, TimerProc) ;
```

如果没有可用的计时器，那么从 SetTimer 返回的 iTimerID 值将为 NULL。

KillTimer 的第一个参数(通常是窗口句柄)也必须为 NULL,计时器 ID 必须是 SetTimer 的返回值:

```
KillTimer (NULL, iTimerID) ;
```

传递给 TimerProc 计时器函数的 hwnd 参数也必须是 NULL。这种设置计时器的方法很少被使用。如果在您的程序中有一系列不同时刻的 SetTimer 调用,而又不希望跟踪您已经用过了哪些计时器 ID,那么使用此方法是很方便的。

既然您已经知道了如何使用 Windows 计时器,就可以开始讨论一组有用的计时器程序了。

8.3 计时器用于时钟

时钟是计时器的最明显的应用,因此让我们来看看两个时钟,一个数字时钟,一个模拟时钟。

8.3.1 构造数字时钟

程序 8-3 所示的 DIGCLOCK 程序,使用类似 LED 的 7 个显示块显示了当前的时间。

程序 8-3　DIGCLOCK 程序

DIGCLOCK.C

```
/*----------------------
     DIGCLOCK.C——Digital Clock
                 (c) Charles Petzold, 1998
  ----------------------*/

#include <windows.h>

#define ID_TIMER    1

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("DigClock") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
```

```
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Digital Clock"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

void DisplayDigit (HDC hdc, int iNumber)
{
     static BOOL fSevenSegment [10][7] = {
                         1, 1, 1, 0, 1, 1, 1,      // 0
                         0, 0, 1, 0, 0, 1, 0,      // 1
                         1, 0, 1, 1, 1, 0, 1,      // 2
                         1, 0, 1, 1, 0, 1, 1,      // 3
                         0, 1, 1, 1, 0, 1, 0,      // 4
                         1, 1, 0, 1, 0, 1, 1,      // 5
                         1, 1, 0, 1, 1, 1, 1,      // 6
                         1, 0, 1, 0, 0, 1, 0,      // 7
                         1, 1, 1, 1, 1, 1, 1,      // 8
                         1, 1, 1, 1, 0, 1, 1 } ;   // 9
     static POINT ptSegment [7][6] = {
```

```
                         7,  6, 11,  2, 31,  2, 35,  6, 31, 10, 11, 10,
                         6,  7, 10, 11, 10, 31,  6, 35,  2, 31,  2, 11,
                        36,  7, 40, 11, 40, 31, 36, 35, 32, 31, 32, 11,
                         7, 36, 11, 32, 31, 32, 35, 36, 31, 40, 11, 40,
                         6, 37, 10, 41, 10, 61,  6, 65,  2, 61,  2, 41,
                        36, 37, 40, 41, 40, 61, 36, 65, 32, 61, 32, 41,
                         7, 66, 11, 62, 31, 62, 35, 66, 31, 70, 11, 70 } ;
     int          iSeg ;
     for (iSeg = 0 ; iSeg < 7 ; iSeg++)
          if (fSevenSegment [iNumber][iSeg])
               Polygon (hdc, ptSegment [iSeg], 6) ;
}

void DisplayTwoDigits (HDC hdc, int iNumber, BOOL fSuppress)
{
     if (!fSuppress || (iNumber / 10 != 0))
          DisplayDigit (hdc, iNumber / 10) ;

     OffsetWindowOrgEx (hdc, -42, 0, NULL) ;
     DisplayDigit (hdc, iNumber % 10) ;
     OffsetWindowOrgEx (hdc, -42, 0, NULL) ;
}

void DisplayColon (HDC hdc)
{
     POINT ptColon [2][4] = { 2, 21, 6, 17, 10, 21, 6, 25,
                              2, 51, 6, 47, 10, 51, 6, 55 } ;

     Polygon (hdc, ptColon [0], 4) ;
     Polygon (hdc, ptColon [1], 4) ;

     OffsetWindowOrgEx (hdc, -12, 0, NULL) ;
}

void DisplayTime (HDC hdc, BOOL f24Hour, BOOL fSuppress)
{
     SYSTEMTIME st ;

     GetLocalTime (&st) ;

     if (f24Hour)
          DisplayTwoDigits (hdc, st.wHour, fSuppress) ;
     else
          DisplayTwoDigits (hdc, (st.wHour %= 12) ? st.wHour : 12, fSuppress) ;

     DisplayColon (hdc) ;
     DisplayTwoDigits (hdc, st.wMinute, FALSE) ;
     DisplayColon (hdc) ;
     DisplayTwoDigits (hdc, st.wSecond, FALSE) ;
```

```
}
LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static BOOL     f24Hour, fSuppress ;
     static HBRUSH   hBrushRed ;
     static int      cxClient, cyClient ;
     HDC             hdc ;
     PAINTSTRUCT     ps ;
     TCHAR           szBuffer [2] ;

     switch (message)
     {
     case WM_CREATE:
          hBrushRed = CreateSolidBrush (RGB (255, 0, 0)) ;
          SetTimer (hwnd, ID_TIMER, 1000, NULL) ;

                                                  // fall through

     case WM_SETTINGCHANGE:
          GetLocaleInfo (LOCALE_USER_DEFAULT, LOCALE_ITIME, szBuffer, 2) ;
          f24Hour = (szBuffer[0] == '1') ;

          GetLocaleInfo (LOCALE_USER_DEFAULT, LOCALE_ITLZERO, szBuffer, 2) ;
          fSuppress = (szBuffer[0] == '0') ;

          InvalidateRect (hwnd, NULL, TRUE) ;
          return 0 ;

     case WM_SIZE:
          cxClient = LOWORD (lParam) ;
          cyClient = HIWORD (lParam) ;
          return 0 ;

     case WM_TIMER:
          InvalidateRect (hwnd, NULL, TRUE) ;
          return 0 ;

     case WM_PAINT:
          hdc = BeginPaint (hwnd, &ps) ;

          SetMapMode (hdc, MM_ISOTROPIC) ;
          SetWindowExtEx (hdc, 276, 72, NULL) ;
          SetViewportExtEx (hdc, cxClient, cyClient, NULL) ;

          SetWindowOrgEx (hdc, 138, 36, NULL) ;
          SetViewportOrgEx (hdc, cxClient / 2, cyClient / 2, NULL) ;
          SelectObject (hdc, GetStockObject (NULL_PEN)) ;
```

```
          SelectObject (hdc, hBrushRed) ;

          DisplayTime (hdc, f24Hour, fSuppress) ;

          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY:
          KillTimer (hwnd, ID_TIMER) ;
          DeleteObject (hBrushRed) ;
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

DIGCLOCK 窗口如图 8-1 所示。

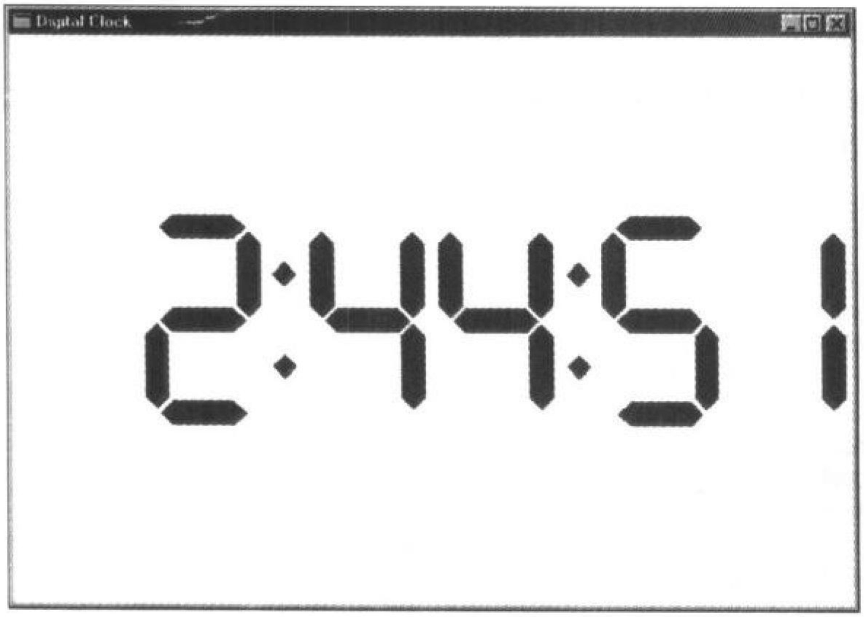


图 8-1 DIGCLOCK 显示

虽然在图 8-1 中您看不到时钟的数字是红色的。DIGCLOCK 的窗口过程在处理 WM_CREATE 消息期间创建了一个红色的刷子并在处理 WM_DESTROY 消息期间清除它。WM_CREATE 消息也为 DIGCLOCK 设置了一个 1 秒的计时器，该计时器在处理 WM_DESTROY 消息期间被终止。(一会儿将讨论对 GetLocaleInfo 的调用。)

在收到 WM_TIMER 消息后，DIGCLOCK 的窗口过程调用 InvalidateRect 简单地使整个窗口无效。这不是最佳方法，因为每秒整个窗口都要被擦除和重画，有时会引起显示

器的闪烁。基于当前的时间使窗口需要更新的部分无效是最好的解决方法。然而,在逻辑上这样做的确很复杂。

在处理 WM_TIMER 消息期间使窗口无效会迫使所有程序的真正活动转入 WM_PAINT。DIGCLOCK 在 WM_PAINT 消息一开始将映射方式设置为 MM_ISOTROPIC。这样,DIGCLOCK 将使用水平方向和垂直方向相等的轴。这些轴(由 SetWindowExtEx 调用设置)是水平 276 个单位,垂直 72 个单位。当然,这些轴定得有点太随意了,但它们是按照时钟数字的大小和间距安排的。

DIGCLOCK 将窗口原点设置为(138,36),这是窗口范围的中心;将视口原点设置为(*cxClient* /2, *cyClient* /2)。这意味着时钟的显示位于 DIGCLOCK 客户区的中心,但是该 DIGCLOCK 也可以使用在显示屏左上角的原点为(0, 0)的轴。

然后 WM_PAINT 将当前刷子设置为早先创建的红刷子,将当前画笔设置为 NULL_PEN, 并调用 DIGCLOCK 中的函数 DisplayTime。

8.3.2 获取当前时间

DisplayTime 函数开始调用 Windows 函数 GetLocalTime,它带有单个的 SYSTEMTIME 结构的参数,在 WINBASE.H 中定义为:

```
typedef struct _SYSTEMTIME
{
     WORD wYear ;
     WORD wMonth ;
     WORD wDayOfWeek ;
     WORD wDay ;
     WORD wHour ;
     WORD wMinute ;
     WORD wSecond ;
     WORD wMilliseconds ;
}
SYSTEMTIME, * PSYSTEMTIME ;
```

很明显,SYSTEMTIME 结构包含日期和时间。月份由 1 开始递增(也就是说,一月是 1),星期由 0 开始递增(星期天是 0)。wDay 成员是本月的当前日子,也是由 1 开始递增的。

SYSTEMTIME 主要用于 GetLocalTime 和 GetSystemTime 函数。GetSystemTime 函数返回当前的世界时间(Coordinated Universal Time, UTC),大概与英国格林威治时间相同。GetLocalTime 函数返回当地时间,依赖于计算机位于的时区。这些值的精确度完全依赖于用户将时间调整的精度以及是否指出了正确的时区。可以双击任务栏的时间显示来检查计算机上的时区设置。第二十三章会有一个程序,能够通过 Internet 精确地设置时间。

Windows 还有 SetLocalTime 和 SetSystemTime 函数,以及在/Platform SDK/Windows Base

Services/General Library/Time 中说明的其他与时间有关的函数。

8.3.3 显示数字和冒号

如果 DIGCLOCK 使用一种模拟 7 段显示的字体会简单一些。否则，它就得使用 Polygon 函数做所有的工作。

DIGCLOCK 中的 DisplayDigit 函数定义了两个数组。fSevenSegment 数组有 7 个 BOOL 值，用于从 0 到 9 的每个十进制数。这些值指出了哪个段需要显示(为 1)，哪个段不需要显示(为 0)。在这个数组中，7 个段由上到下、由左到右排序。7 段中的每个段都是一个 6 边的多边形。ptSegment 数组是一个 POINT 结构的数组，指出了 7 个段中每个点的图形坐标。每个数字由下列代码画出：

```
for (iSeg = 0 ; iSeg < 7 ; iSeg++)
   if (fSevenSegment [iNumber][iSeg])
      Polygon (hdc, ptSegment [iSeg], 6) ;
```

类似地(但更简单)，DisplayColon 函数在小时与分钟、分钟与秒之间画一个冒号。数字是 42 个单位宽，冒号是 12 个单位宽，因此 6 个数字与 2 个冒号，总宽度是 276 个单位，SetWindowExtEx 调用中使用了这个大小。

回到 DisplayTime 函数，原点位于最左数字位置的左上角。DisplayTime 调用 DisplayTwoDigits，DisplayTwoDigits 调用 DisplayDigit 两次，并且在每次调用 OffsetWindowOrgEx 后，将窗口原点向右移动 42 个单位。类似地，DisplayColon 函数在画完冒号后，将窗口原点向右移动 12 个单位。用这种方法，不管对象出现在窗口内的哪个地方，函数对数字和冒号都使用同样的坐标。

这个程序的其他技巧是以 12 小时或 24 小时的格式显示时间以及当最左边的小时数字为 0 时不显示它。

8.3.4 国际化

尽管像 DIGCLOCK 这样显示时间是非常简单的，但是要显示复杂的日期和时间还要依赖 Windows 的国际化支持。格式化日期和时间的最简单的方法是调用 GetDateFormat 和 GetTimeFormat 函数。这些函数在/Platform SDK/Windows Base Services/General Library/String Manipulation/String Manipulation Reference/String Manipulation Functions 中有记载，但是它们在/Platform SDK/Windows Base Services/International Features/National Language Support 中进行了说明。这些函数接受 SYSTEMTIME 结构并且基于用户在“控制面板”的“区域设置”程序中所做的选择而将日期和时间格式化。

DIGCLOCK 不能使用 GetDateFormat 函数，因为它只知道显示数字和冒号，然而，DIGCLOCK 应该能够根据用户的参数选择来显示 12 小时或 24 小时的格式，并禁止(或

不禁止)开头的小时数字。您可以从 GetLocaleInfo 函数中获取这种信息。虽然 GetLocaleInfo 在/Platform SDK/Windows Base Services/General Library/String Manipulation/String Manipulation Reference/String Manipulation Functions 中有记载,但是这个函数使用的标识符在/Platform SDK/Windows Base Services/International Features/National Language Support/National Language Support Constants 中说明。

DIGCLOCK 在处理 WM_CREATE 消息时,最初调用 GetLocaleInfo 两次,第一次使用 LOCALE_ITIME 标识符(确定使用的是 12 小时还是 24 小时格式),然后使用 LOCALE_ITLZERO 标识符(在小时显示中禁止前面显示 0)。GetLocaleInfo 函数在字符串中返回所有的信息,但是在大多数情况下把字符串转变为整数不是非常容易。DIGCLOCK 把字符串保存在两个静态变量中并把它们传递给 DisplayTime 函数。

如果用户更改了任何系统设置,则会将 WM_SETTINGCHANGE 消息传送给所有的应用程序。DIGCLOCK 通过再次调用 GetLocaleInfo 处理这个消息。以这种方式,您可以在"控制面板"的"区域设置"程序中进行不同的设置实验一下。

在理论上,DIGCLOCK 也应该使用 LOCALE_STIME 标识符调用 GetLocaleInfo。这会返回用户为时间的小时、分和秒等单个部分选择的字符。因为 DIGCLOCK 被设置为仅显示冒号,所以即使选择了什么,也会得到冒号。要指出时间是 A.M.或 P.M.,应用程序可以使用带有 LOCALE_S1159 和 LOCALE_S2359 标识符的 GetLocaleInfo 函数。这些标识符使程序获得适合于用户国家/地区和语言的字符串。

我们也可以让 DIGCLOCK 处理 WM_TIMECHANGE 消息,这样它将系统时间与日期发生变化这一消息通知应用程序。DIGCLOCK 因 WM_TIMER 消息而每秒更新一次,实际上没有必要这么做,对 WM_TIMECHANGE 消息的处理使得每分钟更新一次的时钟变得更为合理。

8.3.5 构造模拟时钟

模拟时钟不必关心国际化问题,但是由于图形所引起的复杂性却抵消了这种简化。为了正确地产生时钟,您需要知道一些三角函数。CLOCK 如程序 8-4 所示。

程序 8-4 CLOCK 程序

CLOCK.C

```
/*-----------------------------------
   CLOCK.C—Analog Clock Program
          (c) Charles Petzold, 1998
  -----------------------------------*/
```

```
#include <windows.h>
#include <math.h>

#define ID_TIMER    1
#define TWOPI       (2 * 3.14159)

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("Clock") ;
     HWND         hwnd;
     MSG          msg;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = NULL ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }
     hwnd = CreateWindow (szAppName, TEXT ("Analog Clock"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
```

```
     return msg.wParam ;
}

void SetIsotropic (HDC hdc, int cxClient, int cyClient)
{
     SetMapMode (hdc, MM_ISOTROPIC) ;
     SetWindowExtEx (hdc, 1000, 1000, NULL) ;
     SetViewportExtEx (hdc, cxClient / 2, - cyClient / 2, NULL) ;
     SetViewportOrgEx (hdc, cxClient / 2,   cyClient / 2, NULL) ;
}

void RotatePoint (POINT pt[], int iNum, int iAngle)
{
     int   i ;
     POINT ptTemp ;

     for (i = 0 ; i < iNum ; i++)
     {
          ptTemp.x = (int) (pt[i].x * cos (TWOPI * iAngle / 360) +
               pt[i].y * sin (TWOPI * iAngle / 360)) ;

          ptTemp.y = (int) (pt[i].y * cos (TWOPI * iAngle / 360) -
               pt[i].x * sin (TWOPI * iAngle / 360)) ;

          pt[i] = ptTemp ;
     }
}

void DrawClock (HDC hdc)
{
     int   iAngle ;
     POINT pt[3] ;
     for (iAngle = 0 ; iAngle < 360 ; iAngle += 6)
     {
          pt[0].x =   0 ;
          pt[0].y = 900 ;

          RotatePoint (pt, 1, iAngle) ;

          pt[2].x = pt[2].y = iAngle % 5 ? 33 : 100 ;

          pt[0].x -= pt[2].x / 2 ;
          pt[0].y -= pt[2].y / 2 ;

          pt[1].x = pt[0].x + pt[2].x ;
          pt[1].y = pt[0].y + pt[2].y ;

          SelectObject (hdc, GetStockObject (BLACK_BRUSH)) ;
```

```
          Ellipse (hdc, pt[0].x, pt[0].y, pt[1].x, pt[1].y) ;
     }
}

void DrawHands (HDC hdc, SYSTEMTIME * pst, BOOL fChange)
{
     static POINT pt[3][5] = { 0, -150, 100, 0, 0, 600, -100, 0, 0, -150,
                               0, -200,  50, 0, 0, 800,  -50, 0, 0, -200,
                               0,    0,   0, 0, 0,   0,    0, 0, 0,  800 } ;
     int          i, iAngle[3] ;
     POINT        ptTemp[3][5] ;

     iAngle[0] = (pst->wHour * 30) % 360 + pst->wMinute / 2 ;
     iAngle[1] =  pst->wMinute * 6 ;
     iAngle[2] =  pst->wSecond * 6 ;

     memcpy (ptTemp, pt, sizeof (pt)) ;

     for (i = fChange ? 0 : 2 ; i < 3 ; i++)
     {
          RotatePoint (ptTemp[i], 5, iAngle[i]) ;

          Polyline (hdc, ptTemp[i], 5) ;
     }
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static int         cxClient, cyClient ;
     static SYSTEMTIME  stPrevious ;
     BOOL               fChange ;
     HDC                hdc ;
     PAINTSTRUCT        ps ;
     SYSTEMTIME         st ;

     switch (message)
     {
     case WM_CREATE :
          SetTimer (hwnd, ID_TIMER, 1000, NULL) ;
          GetLocalTime (&st) ;
          stPrevious = st ;
          return 0 ;

     case WM_SIZE :
          cxClient = LOWORD (lParam) ;
          cyClient = HIWORD (lParam) ;
          return 0 ;
```

```
    case WM_TIMER :
         GetLocalTime (&st) ;

         fChange = st.wHour   != stPrevious.wHour ||
                   st.wMinute != stPrevious.wMinute ;

         hdc = GetDC (hwnd) ;

         SetIsotropic (hdc, cxClient, cyClient) ;

         SelectObject (hdc, GetStockObject (WHITE_PEN)) ;
         DrawHands (hdc, &stPrevious, fChange) ;

         SelectObject (hdc, GetStockObject (BLACK_PEN)) ;
         DrawHands (hdc, &st, TRUE) ;

         ReleaseDC (hwnd, hdc) ;

         stPrevious = st ;
         return 0 ;

    case WM_PAINT :
         hdc = BeginPaint (hwnd, &ps) ;

         SetIsotropic (hdc, cxClient, cyClient) ;
         DrawClock    (hdc) ;
         DrawHands    (hdc, &stPrevious, TRUE) ;

         EndPaint (hwnd, &ps) ;
         return 0 ;

    case WM_DESTROY :
         KillTimer (hwnd, ID_TIMER) ;
         PostQuitMessage (0) ;
         return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

CLOCK 屏幕显示如下页图 8-2。

各向同性(isotropic)映射发生对于这样的应用来说是理想的,CLOCK.C 中的 SetIsotropic 函数负责设置此模式。在调用 SetMapMode 之后,SetIsotropic 将窗口范围设置为 1000,并将视口范围设置为客户区的一半宽度和客户区的负的一半高度。视口原点被设置为客户区的中心。在第五章中讨论过,这将创建一个笛卡儿坐标系,其点(0,0)位于客户区的中心,在所有方向上的范围都是 1000。

RotatePoint 函数是用到三角的地方,此函数的三个参数分别是一个或者多个点的数

组、数组中点的个数,以及以度为单位的旋转角度。函数以原点为中心按顺时针方向(这对一个时钟正合适)旋转这些点。例如,如果传给函数的点是(0,100)——即 12:00 的位置——而角度为 90°,那么该点将被变换为(100,0)——即 3:00。它使用下列公式来做到这一点:

$$x' = x^* \cos(a) + y^* \sin(a)$$

$$y' = y^* \cos(a) - x^* \sin(a)$$

RotatePoint 函数在绘制时钟表面的点和表针时都是有用的,我们将马上看到这一点。

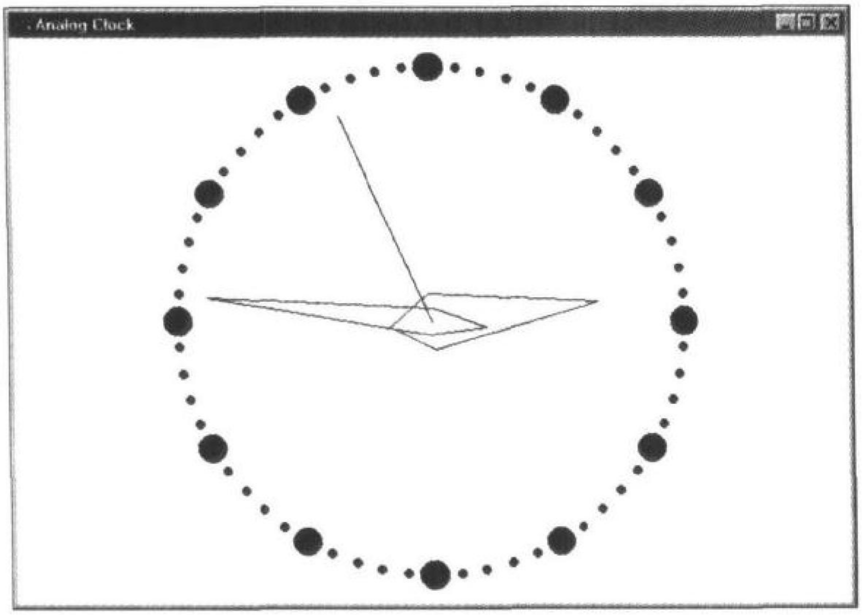


图 8-2 CLOCK 显示

DrawClock 函数绘制 60 个时钟表面的点,从顶部(12:00)开始,其中每个点离原点 900 单位,因此第一个点位于(0,900),此后的每个点按顺时针依次增加 6°。这些点中的 12 个直径为 100 个单位;其余的为 33 个单位。使用 Ellipse 函数来画点。

DrawHands 函数绘制时钟的时针、分针和秒针。定义表针轮廓(当它们垂直向上时的形状)的坐标存放在一个 POINT 结构的数组中。根据时间,这些坐标使用 RotatePoint 函数进行旋转,并用 Windows 的 Polyline 函数进行显示。注意时针和分针只有当传递给 DrawHands 的 bChange 参数为 TRUE 时才被显示。当程序更新时钟的表针时,在多数情况下时针和分针不需要重画。

现在让我们将注意力转到窗口过程。在 WM_CREATE 消息期间,窗口过程获取当前时间,并将它存放在名为 dtPrevious 的变量中,这个变量将在以后被用于确定时针或者分针从上次更新以来是否改变过。

第一次绘制时钟是在第一个 WM_PAINT 消息期间,这只不过是依次调用 SetIsotropic、DrawClock 和 DrawHands,后者的 bChange 参数被设置为 TRUE。

在 WM_TIMER 消息期间,WndProc 首先获取新的时间,并确定是否需要重新绘制时针和分针。如果需要,则使用一个白色画笔和上一次的时间绘制所有的表针,从而有效地擦除它们。否则,只对秒针使用白色画笔进行擦除,然后再使用一个黑色画笔绘制所有的表针。

8.4 为状态报告使用计时器

本章的最后一个程序是我在第五章提到过的。它是一个使用 GetPixel 函数的好例子。

WHATCLR (见程序 8-5)显示了鼠标光标下的当前像素的 RGB 颜色。

程序 8-5 WHATCLR 程序

WHATCLR.C

```
/*---------------------------------------------
     WHATCLR.C——Displays Color Under Cursor
                (c) Charles Petzold, 1998
  ---------------------------------------------*/

#include <windows.h>

#define ID_TIMER    1

void FindWindowSize (int *, int *) ;
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("WhatClr") ;
     HWND         hwnd ;
     int          cxWindow, cyWindow ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
```

```
    wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
    wndclass.lpszMenuName  = NULL ;
    wndclass.lpszClassName = szAppName ;

    if (!RegisterClass (&wndclass))
    {
        MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                    szAppName, MB_ICONERROR) ;
        return 0 ;
    }

    FindWindowSize (&cxWindow, &cyWindow) ;

    hwnd = CreateWindow (szAppName, TEXT ("What Color"),
                         WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_BORDER,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         cxWindow, cyWindow,
                         NULL, NULL, hInstance, NULL) ;

    ShowWindow (hwnd, iCmdShow) ;
    UpdateWindow (hwnd) ;

    while (GetMessage (&msg, NULL, 0, 0))
    {
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
    }
    return msg.wParam ;
}

void FindWindowSize (int * pcxWindow, int * pcyWindow)
{
    HDC        hdcScreen ;
    TEXTMETRIC tm ;

    hdcScreen = CreateIC (TEXT ("DISPLAY"), NULL, NULL, NULL) ;
    GetTextMetrics (hdcScreen, &tm) ;
    DeleteDC (hdcScreen) ;

    * pcxWindow = 2 * GetSystemMetrics (SM_CXBORDER) +
                    12 * tm.tmAveCharWidth ;

    * pcyWindow = 2 * GetSystemMetrics (SM_CYBORDER) +
                  GetSystemMetrics (SM_CYCAPTION) +
                    2 * tm.tmHeight ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
```

```
{
     static COLORREF  cr, crLast ;
     static HDC       hdcScreen ;
     HDC              hdc ;
     PAINTSTRUCT      ps ;
     POINT            pt ;
     RECT             rc ;
     TCHAR            szBuffer [16] ;

     switch (message)
     {
     case WM_CREATE:
          hdcScreen = CreateDC (TEXT ("DISPLAY"), NULL, NULL, NULL) ;
          SetTimer (hwnd, ID_TIMER, 100, NULL) ;
          return 0 ;

     case WM_TIMER:
          GetCursorPos (&pt) ;
          cr = GetPixel (hdcScreen, pt.x, pt.y) ;

          SetPixel (hdcScreen, pt.x, pt.y, 0) ;

          if (cr != crLast)
          {
               crLast = cr ;
               InvalidateRect (hwnd, NULL, FALSE) ;
          }
          return 0 ;

     case WM_PAINT:
          hdc = BeginPaint (hwnd, &ps) ;

          GetClientRect (hwnd, &rc) ;

          wsprintf (szBuffer, TEXT ("  %02X %02X %02X  "),
                    GetRValue (cr), GetGValue (cr), GetBValue (cr)) ;

          DrawText (hdc, szBuffer, -1, &rc,
                    DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;

          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY:
          DeleteDC (hdcScreen) ;
          KillTimer (hwnd, ID_TIMER) ;
          PostQuitMessage (0) ;
          return 0 ;
```

```
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

WHATCLR 在 WinMain 中做了一点与以往不同的事。因为 WHATCLR 的窗口只需要显示十六进制 RGB 值那么大，所以它在 CreateWindow 函数中使用 WS_BORDER 窗口风格创建了一个不能改变大小的窗口。要计算窗口的大小，WHATCLR 通过先调用 CreateIC 再调用 GetSystemMetrics 获取用于视频显示的设备描述表信息。计算的窗口宽度和高度值被传递给 CreateWindow。

WHATCLR 的窗口过程在处理 WM_CREATE 消息期间，调用 CreateDC 创建了用于整个视频显示的设备描述表。这个设备描述表在程序的生存期内都有效。在处理 WM_TIMER 消息期间，程序获取当前鼠标光标位置的像素点。在处理 WM_PAINT 消息期间显示 RGB 颜色。

您可能想知道，从 CreateDC 函数中获取的设备描述表句柄是否能让您在屏幕的任意部位"显示"一些东西而不只是获取像素颜色。回答是可以。一般让一个应用程序在另一个程序中绘制是不好的，但在某些奇怪的情况下，这可能会非常有用。

第九章

子窗口控制

回忆第七章的 CHECKER 程序。这些程序显示了矩形网格。当您在一个矩形中按下鼠标键时,该程序就画一个 x;如果您再按一次鼠标键,那么 x 就消失。虽然这个程序的 CHECKER1 和 CHECKER2 版本只使用一个主窗口,但 CHECKER3 版本却为每个矩形使用一个子窗口。这些矩形由一个叫做 ChildProc 的独立窗口过程维护。

如果有必要,无论矩形是否被选中,那么都可以给 ChildProc 增加一种向其父窗口过程(WndProc)发送消息的手段。通过调用 GetParent,子窗口过程能确定其父窗口的窗口句柄:

```
hwndParent = GetParent (hwnd) ;
```

其中,hwnd 是子窗口的窗口句柄。它可以向其父窗口过程发送消息:

```
SendMessage (hwndParent, message, wParam, lParam) ;
```

那么 message 应该设置为什么呢? 您可以随意地设置,数值大小可以与 WM_USER 相同或更大,这些数字代表和预定义的 WM_messages 不冲突的消息。也许对这个消息,子窗口可以将 wParam 设置为它的子窗口 ID。如果在该子窗口单击,那么 lParam 可以被置为 1;如果未在该子窗口上单击,那么 lParam 被置为 0。这是一种选择。

事实上,这是在建立一个"子窗口控制"。当子窗口的状态已经改变时,子窗口处理鼠标和键盘消息并通知父窗口。使用这种方法,子窗口就变成了其父窗口高一级的输入设备。它将与自己在屏幕上的图形外观相应的处理、它对用户输入的响应,以及在发生一个重要的输入事件时通知另一个窗口的方法封装起来。

虽然您可以建立自己的子窗口控制,但是也可以利用一些预定义的窗口类(和窗口过程)来建立标准的子窗口控制,您一定在别的基于 Windows 的程序中见过了这些控制。这些控制采用的形式有:按钮、复选框、编辑框、列表框、组合框、文本串和滚动条。

例如,如果想在您的电子表格程序的某个角上放置一个标有“Recalculate”的按钮,那么您可以通过调用 CreateWindow 来创建这个按钮。您不必担心鼠标操作、按钮显示操作或当按下该按钮时的自动闪烁操作,这些是由 Windows 内部完成的。您所要做的只是捕获 WM_COMMAND 消息——当按钮被按下时,它通过这一消息通知您的窗口过程。真的这样简单吗?是的,一点不错。

子窗口控制在对话框中最常用。在第十一章中您将会看到,子窗口控制的位置和尺寸是在示例程序的资源描述文中的对话框模板中定义的。但是,您也可以使用预定义的、位于普通窗口客户区表面的子窗口控制。您可以调用一次 CreateWindow 来建立一个子窗口,并通过调用 MoveWindow 来调整子窗口的位置和尺寸。父窗口过程向子窗口控制发送消息,子窗口控制向父窗口过程返回消息。

在建立普通窗口时,首先定义窗口类,并使用 RegisterClass 将其注册到 Windows 中,然后用 CreateWindow 命令在该窗口类的基础上建立一个普通窗口,从第三章开始,我们就是这么做的。但是,当您使用预先定义的某个控制时,不必为子窗口注册窗口类,窗口类已经存在于 Windows 之中,并且有一个预先定义的名字。您只需在 CreateWindow 中把它们用作窗口类参数。CreateWindow 中的窗口风格参数准确地定义了子窗口控制的外形和功能。Windows 包含了处理发送给基于这些窗口类的子窗口的消息的窗口过程。

直接在您的窗口表面使用子窗口控制完成某些任务,这些任务的层次低于在对话框中使用子窗口控制所要求的层次。这里,对话框管理器在您的程序和控制之间增加一个隔离层。值得指出的是,您可能会发现在您的窗口表面创建的子窗口控制没有利用 Tab 键或方向键将输入焦点从一个控制移动到另一个控制的内部功能。子窗口控制能够获得输入焦点,但是获得后,它将不能把输入焦点返回给父窗口。这就是本章要解决的问题。

Windows 编程的文档在两个地方讨论了子窗口控制:每一处,简单的标准控制,我们可以在/Platform SDK/User Interface Services/Controls 的文档所描述的不计数的对话框中看到。这些子窗口包括按钮(其中包括复选框和单选按钮)、静态控制(例如文本标签)、编辑框(您可以在此编辑一行或多行文本)、滚动条、列表框,以及组合框。除组合框以外,在 Windows 1.0 中就包括了这些控制。Windows 文档内容这部分还包括多信息文本编辑控制,它与编辑框相似,但还允许编辑具有不同字体等格式的字本,以及应用程序的桌面工具栏。

相对于“常用控制”,还有一些深奥的特殊控制。这些控制在/Platform SDK/User Interface Services/Shell and Common Controls/Common Controls 描述。本章不讨论常用控制,但它们将出现在本书的其他部分。如果发现 Windows 应用程序中有些部分对您自己的应用程序有用,那么这部分的 Windows 文档很容易找到。

9.1 按钮类

下面我们将通过叫做 BTNLOOK ("button look")的程序来开始介绍按钮窗口类,如程序 9-1 所示。BTNLOOK 创建 10 个子窗口按钮控制,每个控制对应于一个标准的按钮风格,因此共有 10 种标准按钮风格。

程序 9-1 BTNLOOK 程序

BTNLOOK.C

```
/*-----------------------------------------
   BTNLOOK.C——Button Look Program
              (c) Charles Petzold, 1998
  -----------------------------------------*/

#include <windows.h>

struct
{
     int      iStyle ;
     TCHAR *  szText ;
}
button[] =
{
     BS_PUSHBUTTON,        TEXT ("PUSHBUTTON"),
     BS_DEFPUSHBUTTON,     TEXT ("DEFPUSHBUTTON"),
     BS_CHECKBOX,          TEXT ("CHECKBOX"),
     BS_AUTOCHECKBOX,      TEXT ("AUTOCHECKBOX"),
     BS_RADIOBUTTON,       TEXT ("RADIOBUTTON"),
     BS_3STATE,            TEXT ("3STATE"),
     BS_AUTO3STATE,        TEXT ("AUTO3STATE"),
     BS_GROUPBOX,          TEXT ("GROUPBOX"),
     BS_AUTORADIOBUTTON,   TEXT ("AUTORADIO"),
     BS_OWNERDRAW,         TEXT ("OWNERDRAW ")
} ;

#define NUM (sizeof button / sizeof button[0])

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("BtnLook") ;
```

```
     HWND       hwnd ;
     MSG        msg ;
     WNDCLASS   wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Button Look"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static HWND  hwndButton[NUM] ;
     static RECT  rect ;
     static TCHAR szTop[]    = TEXT ("message            wParam       lParam"),
                  szUnd[]    = TEXT ("_______            ______       ______"),
                  szFormat[] = TEXT ("%-16s%04X-%04X    %04X-%04X"),
                  szBuffer[50] ;
```

```
static int      cxChar, cyChar ;
HDC             hdc ;
PAINTSTRUCT ps ;
int             i ;

switch (message)
{
case WM_CREATE :
     cxChar = LOWORD (GetDialogBaseUnits ()) ;
     cyChar = HIWORD (GetDialogBaseUnits ()) ;

     for (i = 0 ; i < NUM ; i++)
          hwndButton[i] = CreateWindow ( TEXT("button"),
                              button[i].szText,
                              WS_CHILD | WS_VISIBLE | button[i].iStyle,
                              cxChar, cyChar * (1 + 2 * i),
                              20 * cxChar, 7 * cyChar / 4,
                              hwnd, (HMENU) i,
                              ((LPCREATESTRUCT) lParam) -> hInstance, NULL) ;
     return 0 ;

case WM_SIZE :
     rect.left   = 24 * cxChar ;
     rect.top    =  2 * cyChar ;
     rect.right  = LOWORD (lParam) ;
     rect.bottom = HIWORD (lParam) ;
     return 0 ;
case WM_PAINT :
     InvalidateRect (hwnd, &rect, TRUE) ;

     hdc = BeginPaint (hwnd, &ps) ;
     SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
     SetBkMode (hdc, TRANSPARENT) ;

     TextOut (hdc, 24 * cxChar, cyChar, szTop, lstrlen (szTop)) ;
     TextOut (hdc, 24 * cxChar, cyChar, szUnd, lstrlen (szUnd)) ;

     EndPaint (hwnd, &ps) ;
     return 0 ;

case WM_DRAWITEM :
case WM_COMMAND :
     ScrollWindow (hwnd, 0, -cyChar, &rect, &rect) ;

     hdc = GetDC (hwnd) ;
     SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
```

```
        TextOut (hdc, 24 * cxChar, cyChar * (rect.bottom / cyChar - 1),
                szBuffer,
                wsprintf (szBuffer, szFormat,
                     message == WM_DRAWITEM ? TEXT ("WM_DRAWITEM") :
                                              TEXT ("WM_COMMAND"),
                     HIWORD (wParam), LOWORD (wParam),
                     HIWORD (lParam), LOWORD (lParam))) ;

        ReleaseDC (hwnd, hdc) ;
        ValidateRect (hwnd, &rect) ;
        break ;

    case WM_DESTROY :
        PostQuitMessage (0) ;
        return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

单击按钮时，按钮就给父窗口过程发送一个 WM_COMMAND 消息，也就是我们所熟悉的 WndProc。BTNLOOK 的 WndProc 将该消息的 wParam 参数和 lParam 参数显示在客户区的右边，如图 9-1 所示。

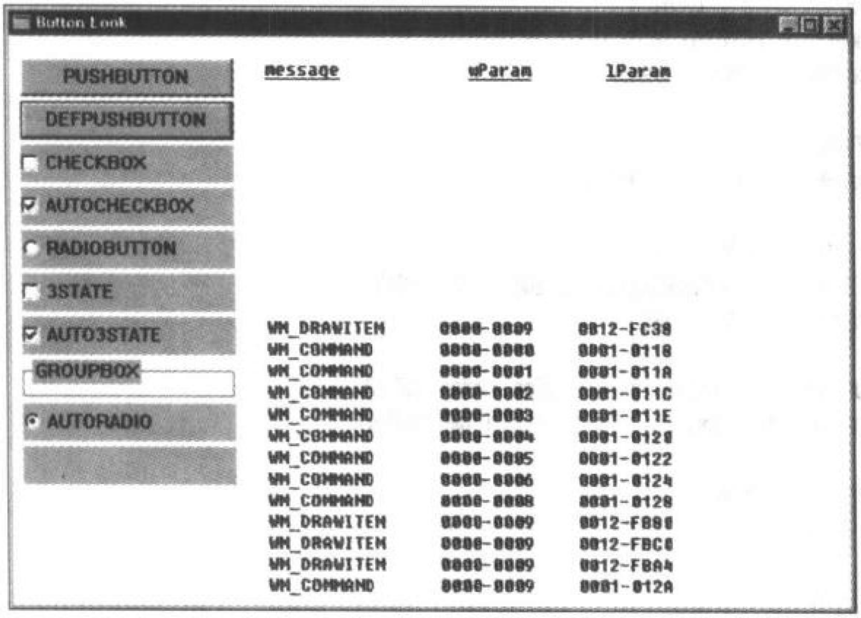


图 9-1　BTNLOOK 的屏幕显示

具有 BS_OWNERDRAW 风格的按钮在窗口上显示为一个背景阴影，因为这种风格的按钮是由程序来负责绘制的。该按钮表示它需要由包含 lParam 消息参数的 WM_

DRAWITEM 消息来绘制，而 lParam 消息参数是一个指向 DRAWITEMSTRUCT 类型结构的指针。在 BTNLOOK 中，这些消息也被同样显示。我将在本章的后面更详细地讨论这种拥有者绘制按钮。

9.1.1 创建子窗口

BTNLOOK 定义了一个叫做 button 的结构，它包括了按钮窗口风格和描述性文本串，它们对应于 10 个按钮类型，所有按钮窗口风格都以字母"BS"开头，它表示"按钮风格"。10 个按钮子窗口是在 WndProc 中处理 WM_CREATE 消息的过程中使用一个 for 循环建立的。CreateWindow 调用使用下面这些参数：

Class name(类名)	TEXT ("button")
Window text(窗口文本)	button[i].szText
Window style(窗口风格)	WS_CHILD | WS_VISIBLE | button[i].iStyle
x position(x 位置)	cxChar
y position(y 位置)	cyChar * (1 + 2 * i)
Width(宽度)	20 * xChar
Height(高度)	7 * yChar / 4
Parent window(父窗口)	hwnd
Child window ID(子窗口 ID)	(HMENU) i
Instance handle(实例句柄)	((LPCREATESTRUCT) lParam) -> hInstance
Extra parameters(额外参数)	NULL

类名参数是预先定义的名字。窗口风格使用 WS_CHILD、WS_VISIBLE，以及在 button 结构中定义的 10 个按钮风格之一（BS_PUSHBUTTON、BS_DEFPUSHBUTTON 等等）。窗口文本参数（对于普通窗口来说，它是显示在标题栏中的文本）将在每个按钮上显示出来。我简单地使用标识按钮风格的文本。

x 位置和 y 位置参数说明子窗口左上角相对于父窗口客户区左上角的位置。宽度和高度参数规定了每个子窗口的宽度的高度。请注意，我用的是 GetDialogBaseUnits 函数来获得默认字体字符的宽度和高度。这是对话框用来获得文本尺寸的函数。此函数返回一个 32 位的值，其中低字表示宽度，高字表示高度。由于 GetDialogBaseUnits 返回的值与从 GetTextMetrics 获得的值基本相同，但 GetDialogBaseUnits 有时使用起来会更方便些，而且能够与对话框控制更好地保持一致。

对每个子窗口，它的子窗口 ID 参数应该各不相同。在处理来自子窗口的 WM_COMMAND 消息时，ID 帮助您的窗口过程识别出相应的子窗口。注意子窗口 ID 是作为 CreateWindow 的一个参数传递的，该参数通常用于指定程序的菜单，因此子窗口 ID 必须

被强制转换为 HMENU。

CreateWindow 调用的实例句柄看起来有点奇怪，但是它利用了如下事实，即在处理 WM_CREATE 消息的过程中，lParam 实际上是指向 CREATESTRUCT（“创建结构”）结构的指针，该结构有一个 hInstance 成员。所以将 lParam 转换成指向 CREATESTRUCT 结构的一个指针，并取出 hInstance。

（有些 Windows 程序使用名为 hInst 的全局变量，使窗口过程能访问 WinMain 中的实例句柄。在 WinMain 中，您只需在建立主窗口之前设置：

```
hInst = hInstance ;
```

在第七章中的 CHECKER3 程序中，我们曾用 GetWindowLong 获得到实例句柄：

```
GetWindowLong (hwnd, GWL_HINSTANCE)
```

这几种方法都是正确的。）

在调用 CreateWindow 之后，我们不必再为这些子窗口做任何事情，由 Windows 中的按钮窗口过程负责维护它们，并处理所有重画工作。（BS_OWNERDRAW 风格的按钮例外，它要求程序绘制它，这些将在后面加以讨论。）在程序终止时，如果父窗口已经被清除，那么 Windows 将清除这些子窗口。

9.1.2 子窗口向父窗口发送消息

当您运行 BTNLOOK 时，将看到在客户区的左边会显示出不同的按钮类型。我在前面已经提到过，用鼠标单击按钮时，子窗口控制就向其父窗口发送一个 WM_COMMAND 消息。BTNLOOK 捕获 WM_COMMAND 消息并显示 wParam 和 lParam 的值，它们的含义如下：

LOWORD (wParam)	子窗口 ID
HIWORD (wParam)	通知码
lParam	子窗口句柄

如果您正在将 16 位 Windows 程序移植过来，那么注意改变这些消息参数以容纳 32 位的句柄。

子窗口 ID 是在创建子窗口时传递给 CreateWindow 的值。在 BTNLOOK 中，这些 ID 被显示在客户区中，并使用 0 到 9 分别标识 10 个按钮。子窗口句柄是 Windows 从 CreateWindow 返回的值。

通知码更详细表明了消息的含义。按钮通知码的可能值在 Windows 头文件中定义，如表 9-1 所示。

表 9-1

按钮通知码标识符	值
BN_CLICKED	0
BN_PAINT	1
BN_HILITE or BN_PUSHED	2
BN_UNHILITE or BN_UNPUSHED	3
BN_DISABLE	4
BN_DOUBLECLICKED or BN_DBLCLK	5
BN_SETFOCUS	6
BN_KILLFOCUS	7

实际上,您不会看到这些按钮值中的大多数。从 1 到 4 的通知码用于一种叫做 BS_USERBUTTON 的已不再使用的按钮。(它已经由 BS_OWNERDRAW 和另一种不同的通知方式所替换。)通知码 6 到 7 只有当按钮风格包括标识 BS_NOTIFY 时才发送。通知码 5 只对 BS_RADIOBUTTON、BS_AUTORADIOBUTTON 和 BS_OWNERDRAW 按钮发送,或者当按钮风格中包括 BS_NOTIFY 时,也为其他按钮发送。

您会注意到,当用鼠标单击按钮时,该按钮文本的周围会有虚线。这表示该按钮拥有了输入焦点,所有键盘输入都将传送给子窗口按钮控制,而不是传送给主窗口。然而,当该按钮控制拥有输入焦点时,它将忽略所有的键盘输入,只有 Spacebar 键例外,此时 Spacebar 键与鼠标具有相同的效果。

9.1.3 父窗口向子窗口发送消息

虽然 BTNLOOK 中没有演示这一事实,但是父窗口过程也能向子窗口控制发送消息。这些消息包括以前缀 WM 开头的许多消息。另外,在 WINUSER.H 中还定义了 8 个按钮说明消息;前缀 BM 表示"按钮消息"。这些按钮消息如表 9-2 所示。

表 9-2

按钮消息	值
BM_GETCHECK	0x00F0
BM_SETCHECK	0x00F1
BM_GETSTATE	0x00F2
BM_SETSTATE	0x00F3
BM_SETSTYLE	0x00F4
BM_CLICK	0x00F5
BM_GETIMAGE	0x00F6
BM_SETIMAGE	0x00F7

BM_GETCHECK 和 BM_SETCHECK 消息由父窗口发送给子窗口控制，以获取或者设置复选框和单选按钮的选中标记。BM_GETSTATE 和 BM_SETSTATE 消息表示按钮处于正常状态还是（鼠标或 Spacebar 键按下时的）“按下”状态。我们将在讨论按钮的每种类型时，看到这些消息是如何起作用的。BM_SETSTYLE 消息允许您在按钮创建之后改变按钮风格。

每个子窗口控制都具有一个在其兄弟中唯一的窗口句柄和 ID 值。对于句柄和 ID 这两者，知道其中的一个您就可以获得另一个。如果您知道子窗口控制的窗口句柄，那么您可以用下面的语句来获得 ID：

```
id = GetWindowLong (hwndChild, GWL_ID) ;
```

第七章的 CHECKER3 程序曾用此函数（与 SetWindowLong 一起）来维护注册窗口类时保留的特殊区的数据。在创建子窗口时，Windows 保留了 GWL_ID 标识符访问的数据。您也可以使用

```
id = GetDlgCtrlID (hwndChild) ;
```

虽然函数中的“Dlg”部分指的是对话框，但实际上这是一个通用的函数。

知道 ID 和父窗口句柄，您就能获得子窗口句柄：

```
hwndChild = GetDlgItem (hwndParent, id) ;
```

9.1.4 下压按钮

在 BTNLOOK 中显示的前两个按钮是“下压”按钮。按钮是一个矩形，包括了 CreateWindow 调用中窗口文本参数所指定的文本。该矩形占用了在 CreateWindow 或者 MoveWindow 调用中给出的全部高度和宽度，而文本在矩形的中心。

下压按钮控制主要用来触发一个立即响应的动作，而不保留任何形式的开/关指示。两种类型的按钮控制有两种窗口风格，分别叫做 BS_PUSHBUTTON 和 BS_DEFPUSHBUTTON，BS_DEFPUSHBUTTON 中的“DEF”代表“默认”。当用来设计对话框时，BS_PUSHBUTTON 控制和 BS_DEFPUSHBUTTON 控制的作用不同。但是当用作子窗口控制时，两种类型的按钮作用相同，尽管 BS_DEFPUSHBUTTON 的边框要粗一些。

当按钮的高度为文本字符高度的 7/4 倍时，按钮的外观最好，其中文本字符由 BTNLOOK 使用；按钮的宽度至少调节到文本的宽度再加上两个字符的宽度。

当鼠标光标在按钮中时，按下鼠标键将使按钮用三维阴影重画自己，就好像真地被按下一样。放开鼠标键时，就恢复按钮的原样，并向父窗口发送一个 WM_COMMAND 消息和 BN_CLICKED 通知码。与其他按钮类型相似，当按钮拥有输入焦点时，在文本的周围就有虚线，按下和释放 Spacebar 键与按下和释放鼠标键具有相同的效果。

您可以通过给窗口发送 BM_SETSTATE 消息来模拟按钮闪动。如下操作导致按钮被按下：

```
SendMessage (hwndButton, BM_SETSTATE, 1, 0) ;
```

下面的调用使按钮恢复正常：

```
SendMessage (hwndButton, BM_SETSTATE, 0, 0) ;
```

hwndButton 窗口句柄是从 CreateWindow 调用返回的值。

您也可以向下压按钮发送 BM_GETSTATE 消息，子窗口控制返回按钮的当前状态：如果按钮被按下，则返回 TRUE；如果按钮处于正常状态，则返回 FALSE。但是，绝大多数应用不需要这一消息。因为按钮不保留任何开/关信息，所以 BM_SETCHECK 消息和 BM_GETCHECK 消息不会被用到。

9.1.5 复选框

复选框是一个文本框，文本通常出现在复选框的右边。（如果您在创建按钮时指定了 BS_LEFTTEXT 风格，那么文本会出现在左边；您也许将用 BS_RIGHT 直接调整文本来组合此风格。）复选框通常用于允许用户对选项进行选择的应用程序中。复选框的常用功能如同一个开关：单击框一次将显示复选标记，再次单击清除复选标记。

复选框最常用的两种风格是 BS_CHECKBOX 和 BS_AUTOCHECKBOX。在使用 BS_CHECKBOX 时，您需要自己向该控制发送 BM_SETCHECK 消息来设置复选标记。WParam 参数置 1 时设置复选标记，置 0 时清除复选标记。通过向该控制发送 BM_GETCHECK 消息，您可以得到该复选框的当前状态。在处理来自控制的 WM_COMMAND 消息时，您可以用如下的指令来翻转 X 标记：

```
SendMessage ((HWND) lParam, BM_SETCHECK, (WPARAM)
        !SendMessage ((HWND) lParam, BM_GETCHECK, 0, 0), 0) ;
```

注意第二个 SendMessage 调用前面的操作符“!”，其中 lParam 是在 WM_COMMAND 消息中传给用户窗口过程的子窗口句柄。如果您以后又想知道按钮的状态，那么可以向它发送另一个 BM_GETCHECK 消息；您也可以将当前状态保存在您的窗口过程中的一个静态变量里，或者向它发送 BM_SETCHECK 消息来初始化带复选标记的 BS_CHECKBOX 复选框：

```
SendMessage (hwndButton, BM_SETCHECK, 1, 0) ;
```

对 BS_AUTOCHECKBOX 风格，按钮自己触发复选标记的开和关，您的窗口过程可

以忽略 WM_COMMAND 消息。当您需要按钮的当前状态时，可以向控制发送 BM_GET-CHECK 消息：

```
iCheck = (int) SendMessage (hwndButton, BM_GETCHECK, 0, 0) ;
```

如果该按钮被选中，则 iCheck 的值为 TRUE 或者非零数；如果按钮未被选中，则 iCheck 的值为 FALSE 或者 0。

其余两种复选框风格是 BS_3STATE 和 BS_AUTO3STATE，正如它们的名字所暗示的，这两种风格能显示第三种状态——复选框内是灰色——它出现在向控制发送 wParam = 2 的 WM_SETCHECK 消息时。灰色是向用户表明此框是不能被选中的或者是禁用的。

复选框沿矩形的左边框对齐，并集中在调用 CreateWindow 过程中规定的矩形的顶边和底边之间，在该矩形内的任何地方按下鼠标都会向其父窗口发送一个 WM_COMMAND 消息。复选框的最小高度是一个字符的高度，最小宽度是文本中的字符数加 2。

9.1.6 单选按钮

单选按钮的名称在一列按钮的后面，这些按钮就像汽车上的收音机一样。汽车收音机上的每一个按钮都对应一种收音状态，而且一次只能有一个按钮被按下。在对话框中，单选按钮组常常用来表示相互排斥的选项。与复选框不同，单选按钮的工作与开关不一样，也就是说，当第二次按单选按钮时，它的状态保持不变。

单选按钮的形状是一个圆圈，而不是方框，除此之外，它非常像复选框。圆圈内的加重圆点表示该单选按钮已经被选中。单选按钮有窗口风格 BS_RADIOBUTTON 或 BS_AUTORADIOBUTTON 两种，但是后者只用于对话框。

当您收到来自单选按钮的 WM_COMMAND 消息时，应该向它发送 wParam 等于 1 的 BM_SETCHECK 消息来显示其选中状态：

```
SendMessage (hwndButton, BM_SETCHECK, 1, 0) ;
```

对相同组中的其他所有单选按钮，您可以通过向它们发送 wParam 等于 0 的 BM_SET-CHECK 消息来显示其未选中状态：

```
SendMessage (hwndButton, BM_SETCHECK, 0, 0) ;
```

9.1.7 分组框

分组框即风格为 BS_GROUPBOX 的选择框，它是按钮类中的特例，既不处理鼠标输入和键盘输入，也不向其父窗口发送 WM_COMMAND 消息。分组框是一个矩形框，窗口文

本在其顶部显示。分组框常用来包含其他的按钮控制。

9.1.8 更改按钮文本

您可以通过调用下面的 SetWindowText 来更改按钮(或者其他任何窗口)内的文本:

```
SetWindowText (hwnd, pszString) ;
```

其中 hwnd 是要更改的窗口的句柄,pszString 是一个指向 null 终结串的指针。对于一般的窗口来说,这个文本是标题栏的文本;对于按钮控制来说,它是随该按钮显示的文本。

您也可以获取窗口的当前文本:

```
iLength = GetWindowText (hwnd, pszBuffer, iMaxLength) ;
```

iMaxLength 指定复制到 pszBuffer 指向的缓冲区中的最大字符数,该函数返回复制的字符数。您可以首先通过调用

```
iLength = GetWindowTextLength (hwnd) ;
```

来获得特定文本的长度。

9.1.9 可见的和启用的按钮

为了接收鼠标和键盘输入,子窗口必须是可见的(被显示)和被启用的。当窗口是可见的而未被启用时,那么窗口以灰色而非黑色显示正文。

如果在建立子窗口时,您没有将 WS_VISIBLE 包含在窗口类中,那么直到调用

```
ShowWindow (hwndChild, SW_SHOWNORMAL) ;
```

时子窗口才被显示出来。

如果您将 WS_VISIBLE 包含在窗口类中,就没有必要调用 ShowWindow。但是,您可以通过调用

```
ShowWindow (hwndChild, SW_HIDE) ;
```

将子窗口隐藏起来。

您可以通过下面的调用来确定子窗口是否可见:

```
IsWindowVisible (hwndChild) ;
```

您也可以使子窗口被启用或者不被启用。默认情况下,窗口是被启用的。您可以通过调用

```
EnableWindow (hwndChild, FALSE) ;
```

使窗口不被启用。

对于按钮控制，这具有使按钮文本串变灰的作用。按钮将不再对鼠标输入和键盘输入做出响应，这是表示按钮选项当前不可用的最好方法。

您可以通过调用

```
EnableWindow (hwndChild, TRUE) ;
```

使子窗口再次被启用。

您还可以通过调用

```
IsWindowEnabled (hwndChild) ;
```

来确定子窗口是否被启用。

9.1.10 按钮和输入焦点

在本章前面已经提到过，当用鼠标单击按钮、复选框、单选框和拥有者绘制按钮时，它们接收到输入焦点。这些控制使用文本周围的虚线来表示它拥有了输入焦点。当子窗口控制得到输入焦点时，其父窗口就失去了输入焦点；所有的键盘输入都进入子窗口控制中，而不会进入父窗口中。但是，子窗口控制只对 Spacebar 键做出响应，此时 Spacebar 键的作用就如同鼠标键一样。这种情形导致了一个明显的问题：您的程序失去了对键盘处理的控制。让我们看看对此我们能做一些什么。

在第六章中已经提到过，当 Windows 将输入焦点从一个窗口（例如一个父窗口）转换到另一个窗口（例如一个子窗口控制）时，它首先给正在失去输入焦点的窗口发送一个 WM_KILLFOCUS 消息，wParam 参数是接收输入焦点的窗口的句柄。然后，Windows 向正在接收输入焦点的窗口发送一个 WM_SETFOCUS 消息，同时 wParam 是还在失去输入焦点的窗口的句柄。（在这两种情况中，wParam 可能取值 NULL，它表示没有窗口拥有或者正在接收输入焦点。）

通过处理 WM_KILLFOCUS 消息，父窗口可以阻止子窗口控制获得输入焦点。假定数组 hwndChild 包含了所有子窗口的窗口句柄。（它们是在调用 CreateWindow 来建立窗口的时候保存到数组中的。）NUM 是子窗口的数目：

```
case WM_KILLFOCUS :
    for (i = 0 ; i < NUM ; i++)
        if (hwndChild [i] == (HWND) wParam)
        {
            SetFocus (hwnd) ;
            break ;
        }
    return 0 ;
```

在这段代码中，当父窗口获知它正在失去输入焦点而让它的某个子窗口得到输入焦点

时，它将调用 SetFocus 来恢复输入焦点给自己所有。

下面是达到相同目的的更简单(但不太直观)的方法：

```
case WM_KILLFOCUS :
    if (hwnd == GetParent ((HWND) wParam))
        SetFocus (hwnd) ;
    return 0 ;
```

但是，这两种方法都有缺点：它们阻止按钮对 Spacebar 键做出响应，因为该按钮总是得不到输入焦点。一个更好的方法是使按钮得到输入焦点，也能让用户用 Tab 键从一个按钮转移到另一个按钮。这听起来似乎不可能，但是在本章的后面，我们将要说明在 COLORS1 程序中如何用“窗口子类化”技术来实现这种方法。

9.2 控制与颜色

您可以在图 9-1 中看到，许多按钮的显示看起来并不正确。下压按钮还好，但是其他按钮却带有一个本不应该在那里的矩形灰色背景。这是因为这些按钮本来是为对话框中的显示而设计的，而在 Windows 98 中，对话框有一个灰色的表面。我们的窗口有一个白色的表面，这是因为我们在 WNDCLASS 结构

```
wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
```

中就是这样定义的。

我们已经这么做了，因为我们经常在客户区中显示文本，而 GDI 使用在默认设备描述表中定义的文本颜色和背景颜色，它们总是黑色和白色。为了使这些按钮更加美观一些，我们必须要么改变客户区的颜色使之和按钮的背景颜色一致，要么以某种方法将按钮的背景颜色改为白色。

解决此问题的第一步是理解 Windows 对“系统颜色”的使用。

9.2.1 系统颜色

Windows 保留了 29 种系统颜色以供各种显示使用。您可以使用 GetSysColor 和 SetSysColors 来获得和设置这些颜色。在 Windows 头文件中定义的标识符规定了系统颜色。使用 SetSysColors 设置的系统颜色只在当前 Windows 会话过程中有效。

借助 Windows“控制面板”程序的“显示”部分，您可以改变一些(但不是全部)系统颜色。若为 Microsoft Windows NT，选中的颜色保存在注册表中；若为 Microsoft Windows 98，则保存在 WIN.INI 文件中。注册表和 WIN.INI 文件都为这 29 种系统颜色使用了关键字(与 GetSysColor 和 SetSysColors 的标识符不同)，在系统颜色的后面跟着红、绿、蓝三种颜色的值，该值的变化范围是 0 到 255。表 9-3 说明了这 29 种系统颜色是如何在

GetSysColor、SetSysColors 以及 WIN.INI 关键字中用常数来标识的。表 9-3 是按 COLOR_常数值(以 0 开始到 28 结束)顺序排列的。这 29 种颜色的默认值是由显示驱动程序提供的,在不同的机器上可能略有不同。

表 9-3

GetSysColor 和 SetSysColors	注册表键或 WIN.INI 标识符	默认的 RGB 值
COLOR_SCROLLBAR	Scrollbar	C0-C0-C0
COLOR_BACKGROUND	Background	00-80-80
COLOR_ACTIVECAPTION	ActiveTitle	00-00-80
COLOR_INACTIVECAPTION	InactiveTitle	80-80-80
COLOR_MENU	Menu	C0-C0-C0
COLOR_WINDOW	Window	FF-FF-FF
COLOR_WINDOWFRAME	WindowFrame	00-00-00
COLOR_MENUTEXT	MenuText	C0-C0-C0
COLOR_WINDOWTEXT	WindowText	00-00-00
COLOR_CAPTIONTEXT	TitleText	FF-FF-FF
COLOR_ACTIVEBORDER	ActiveBorder	C0-C0-C0
COLOR_INACTIVEBORDER	InactiveBorder	C0-C0-C0
COLOR_APPWORKSPACE	AppWorkspace	80-80-80
COLOR_HIGHLIGHT	Highlight	00-00-80
COLOR_HIGHLIGHTTEXT	HighlightText	FF-FF-FF
COLOR_BTNFACE	ButtonFace	C0-C0-C0
COLOR_BTNSHADOW	ButtonShadow	80-80-80
COLOR_GRAYTEXT	GrayText	80-80-80
COLOR_BTNTEXT	ButtonText	00-00-00
COLOR_INACTIVECAPTIONTEXT	InactiveTitleText	C0-C0-C0
COLOR_BTNHIGHLIGHT	ButtonHighlight	FF-FF-FF
COLOR_3DDKSHADOW	ButtonDkShadow	00-00-00
COLOR_3DLIGHT	ButtonLight	C0-C0-C0
COLOR_INFOTEXT	InfoText	00-00-00
COLOR_INFOBK	InfoWindow	FF-FF-FF
[no identifier; use value 25]	ButtonAlternateFace	B8-B4-B8
COLOR_HOTLIGHT	HotTrackingColor	00-00-FF
COLOR_GRADIENTACTIVECAPTION	GradientActiveTitle	00-00-80
COLOR_GRADIENTINACTIVECAPTION	GradientInactiveTitle	80-80-80

坏消息:虽然这些颜色中的许多似乎都是不言自明的(例如,COLOR_BACKGROUND 是所有窗口后面的桌面区域的颜色),在近版的 Windows 中系统颜色的使用变得非常混乱。以前,Windows 在视觉上要比今天简单得多。实际上,在 Windows 3.0 以

前，只定义了前 13 种系统颜色。随着看起来越来越难于控制的三维外观的使用，相应地也需要越来越多的系统颜色。

9.2.2 按钮颜色

对每一个需要多种颜色的按钮来说，这个问题就更加明显。COLOR_BTNFACE 被用于下压按钮主要的表面颜色以及其他按钮主要的背景颜色。（这也是用于对话框和消息框的系统颜色。）COLOR_BTNSHADOW 被建议用作下压按钮右下边以及复选框内部和单选按钮圆点的阴影。对于下压按钮，COLOR_BTNTEXT 被用作文本颜色；而对于其他的按钮，则使用 COLOR_WINDOWTEXT 作为文本颜色。还有其他几种系统颜色用于按钮设计的各个部分。

因此，如果在您想在我们的客户区表面显示按钮，那么一种避免颜色冲突的方法是屈服于这些系统颜色。首先，在定义窗口类时使用 COLOR_BTNFACE 作为您的客户区的背景颜色：

```
wndclass.hbrBackground = (HBRUSH) (COLOR_BTNFACE + 1) ;
```

您可以在 BTNLOOK 程序中尝试这种方法。当 WNDCLASS 结构中的 hbrBackground 值是这个值时，Windows 会明白这实际上指的是一种系统颜色而非一个实际的句柄。Windows 要求当您在 WNDCLASS 结构的 hbrBackground 域中指定这些标识符时加 1，这样做的目的是防止其值为 NULL，而没有任何其他目的。如果在您的程序运行中，系统颜色恰好发生了变化，那么客户区将变得无效，Windows 将使用新的 COLOR_BTNFACE 值。但是现在我们又引发了另一个问题。当您使用 TextOut 显示文本时，Windows 使用的是在设备描述表中为背景颜色（它擦除文本后的背景）和文本颜色定义的值，其默认值为白色（背景）和黑色（文本），而不管系统颜色和窗口类结构中的 hbrBackground 域为何值。所以，您需要使用 SetTextColor 和 SetBkColor 将文本和文本背景的颜色改变为系统颜色。您可以在获得设备描述表句柄之后这么做：

```
SetBkColor (hdc, GetSysColor (COLOR_BTNFACE)) ;
SetTextColor (hdc, GetSysColor (COLOR_WINDOWTEXT)) ;
```

这样，客户区背景、文本背景和文本的颜色都与按钮的颜色一致了。但是，如果当您的程序运行时，用户改变了系统颜色，您可能要改变文本背景颜色和文本颜色。这时您可以使用下面的代码：

```
case WM_SYSCOLORCHANGE:
    InvalidateRect (hwnd, NULL, TRUE) ;
    break ;
```

9.2.3 WM_CTLCOLORBTN 消息

我们已经看到了是如何将客户区的颜色和文本颜色调节成按钮的背景颜色的。我们是否可以将程序中按钮的颜色调节为喜欢的颜色呢？理论上没问题，但在实践中别这样做。用 SetSysColors 来改变按钮的外观可能不是您想做的，这会影响当前在 Windows 下运行的所有程序；也是用户不太喜欢的。

更好的方法(同样也只是理论上的)是处理 WM_CTLCOLORBTN 消息，这是当子窗口即将给其客户区着色时，由按钮控制发送给其父窗口过程的一个消息。父窗口可以利用这个机会来改变子窗口过程将要用来着色的颜色。(在 Windows 的 16 位版本中，一个称为 WM_CTLCOLOR 的消息被用于所有的控制，现在针对每种类型的标准控制，分别代之以不同的消息。)

当父窗口过程收到 WM_CTLCOLORBTN 消息时，wParam 消息参数是按钮的设备描述表句柄，lParam 是按钮的窗口句柄。当父窗口过程得到这个消息时，按钮控制已经获得了它的设备描述表。当您的窗口过程处理一个 WM_CTLCOLORBTN 消息时，您必须完成如下三个动作：

- 使用 SetTextColor 有选择地设置一种文本颜色；
- 使用 SetBkColor 有选择地设置一种文本背景颜色；
- 将一个画刷句柄返回给子窗口。

理论上，子窗口使用这个画刷来着色背景。当您不再需要这个画刷时，您应该负责将它清除。

下面是使用 WM_CTLCOLORBTN 的问题所在：只有下压按钮和拥有者绘制按钮才给其父窗口发送 WM_CTLCOLORBTN，而只有拥有者绘制按钮才响应父窗口过程对消息的处理，使用画刷来着色背景。这基本上是没有意义的，因为无论怎样都是由父窗口来负责绘制拥有者绘制按钮。

在本章后面，我们将说明，在某些情况下，一些类似于 WM_CTLCOLORBTN 但适用于其他类型控制的消息更为有用。

9.2.4 拥有者绘制按钮

如果您想对按钮的全部可见的外观部分实行全面的控制，但又不希望受键盘和鼠标逻辑的影响，那么您可以创建一个 BS_OWNERDRAW 风格的按钮，具体情况如程序 9-2 所演示的那样。

程序 9-2 OWNDRAW 程序

OWNDRAW.C

```
/*-------------------------------------------------------
   OWNDRAW.C——Owner-Draw Button Demo Program
              (c) Charles Petzold, 1996
  -------------------------------------------------------*/
#include <windows.h>

#define ID_SMALLER     1
#define ID_LARGER      2
#define BTN_WIDTH      (8 * cxChar)
#define BTN_HEIGHT     (4 * cyChar)

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

HINSTANCE hInst ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("OwnDraw") ;
     MSG          msg ;
     HWND         hwnd ;
     WNDCLASS     wndclass ;

     hInst = hInstance ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = szAppName ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }
```

```
     hwnd = CreateWindow (szAppName, TEXT ("Owner-Draw Button Demo"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

void Triangle (HDC hdc, POINT pt[])
{
     SelectObject (hdc, GetStockObject (BLACK_BRUSH)) ;
     Polygon (hdc, pt, 3) ;
     SelectObject (hdc, GetStockObject (WHITE_BRUSH)) ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static HWND       hwndSmaller, hwndLarger ;
     static int        cxClient, cyClient, cxChar, cyChar ;
     int               cx, cy ;
     LPDRAWITEMSTRUCT  pdis ;
     POINT             pt[3] ;
     RECT              rc ;

     switch (message)
     {
     case WM_CREATE :
          cxChar = LOWORD (GetDialogBaseUnits ()) ;
          cyChar = HIWORD (GetDialogBaseUnits ()) ;

               // Create the owner-draw pushbuttons

          hwndSmaller = CreateWindow (TEXT ("button"), TEXT (""),
                                      WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
                                      0, 0, BTN_WIDTH, BTN_HEIGHT,
                                      hwnd, (HMENU) ID_SMALLER, hInst, NULL) ;

          hwndLarger = CreateWindow (TEXT ("button"), TEXT (""),
```

```
                              WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
                              0, 0, BTN_WIDTH, BTN_HEIGHT,
                              hwnd, (HMENU) ID_LARGER, hinst, NULL) ;
          return 0 ;

     case WM_SIZE :
          cxClient = LOWORD (lParam) ;
          cyClient = HIWORD (lParam) ;
               // Move the buttons to the new center

          MoveWindow (hwndSmaller, cxClient / 2 - 3 * BTN_WIDTH / 2,
                                   cyClient / 2 -     BTN_HEIGHT / 2,
                      BTN_WIDTH, BTN_HEIGHT, TRUE) ;
          MoveWindow (hwndLarger, cxClient / 2 +      BTN_WIDTH / 2,
                                  cyClient / 2 -      BTN_HEIGHT / 2,
                      BTN_WIDTH, BTN_HEIGHT, TRUE) ;
          return 0 ;

     case WM_COMMAND :
          GetWindowRect (hwnd, &rc) ;

               // Make the window 10% smaller or larger

          switch (wParam)
          {
          case ID_SMALLER :
               rc.left   += cxClient / 20 ;
               rc.right  -= cxClient / 20 ;
               rc.top    += cyClient / 20 ;
               rc.bottom -= cyClient / 20 ;
               break ;

          case ID_LARGER :
               rc.left   -= cxClient / 20 ;
               rc.right  += cxClient / 20 ;
               rc.top    -= cyClient / 20 ;
               rc.bottom += cyClient / 20 ;
               break ;
          }

          MoveWindow (hwnd, rc.left, rc.top, rc.right - rc.left,
                            rc.bottom - rc.top, TRUE) ;
          return 0 ;

     case WM_DRAWITEM :
          pdis = (LPDRAWITEMSTRUCT) lParam ;
```

```
     // Fill area with white and frame it black

FillRect (pdis -> hDC, &pdis -> rcItem,
       (HBRUSH) GetStockObject (WHITE_BRUSH)) ;

FrameRect (pdis -> hDC, &pdis -> rcItem,
         (HBRUSH) GetStockObject (BLACK_BRUSH)) ;
     // Draw inward and outward black triangles
cx = pdis -> rcItem.right  - pdis -> rcItem.left ;
cy = pdis -> rcItem.bottom - pdis -> rcItem.top ;

switch (pdis -> CtlID)
{
case ID_SMALLER :
     pt[0].x = 3 * cx / 8 ;  pt[0].y = 1 * cy / 8 ;
     pt[1].x = 5 * cx / 8 ;  pt[1].y = 1 * cy / 8 ;
     pt[2].x = 4 * cx / 8 ;  pt[2].y = 3 * cy / 8 ;

     Triangle (pdis -> hDC, pt) ;

     pt[0].x = 7 * cx / 8 ;  pt[0].y = 3 * cy / 8 ;
     pt[1].x = 7 * cx / 8 ;  pt[1].y = 5 * cy / 8 ;
     pt[2].x = 5 * cx / 8 ;  pt[2].y = 4 * cy / 8 ;

     Triangle (pdis -> hDC, pt) ;

     pt[0].x = 5 * cx / 8 ;  pt[0].y = 7 * cy / 8 ;
     pt[1].x = 3 * cx / 8 ;  pt[1].y = 7 * cy / 8 ;
     pt[2].x = 4 * cx / 8 ;  pt[2].y = 5 * cy / 8 ;

     Triangle (pdis -> hDC, pt) ;

     pt[0].x = 1 * cx / 8 ;  pt[0].y = 5 * cy / 8 ;
     pt[1].x = 1 * cx / 8 ;  pt[1].y = 3 * cy / 8 ;
     pt[2].x = 3 * cx / 8 ;  pt[2].y = 4 * cy / 8 ;

     Triangle (pdis -> hDC, pt) ;
     break ;

case ID_LARGER :
     pt[0].x = 5 * cx / 8 ;  pt[0].y = 3 * cy / 8 ;
     pt[1].x = 3 * cx / 8 ;  pt[1].y = 3 * cy / 8 ;
     pt[2].x = 4 * cx / 8 ;  pt[2].y = 1 * cy / 8 ;

     Triangle (pdis -> hDC, pt) ;

     pt[0].x = 5 * cx / 8 ;  pt[0].y = 5 * cy / 8 ;
     pt[1].x = 5 * cx / 8 ;  pt[1].y = 3 * cy / 8 ;
```

```
                pt[2].x = 7 * cx / 8 ;  pt[2].y = 4 * cy / 8 ;

                Triangle (pdis -> hDC, pt) ;
                pt[0].x = 3 * cx / 8 ;  pt[0].y = 5 * cy / 8 ;
                pt[1].x = 5 * cx / 8 ;  pt[1].y = 5 * cy / 8 ;
                pt[2].x = 4 * cx / 8 ;  pt[2].y = 7 * cy / 8 ;

                Triangle (pdis -> hDC, pt) ;
                pt[0].x = 3 * cx / 8 ;  pt[0].y = 3 * cy / 8 ;
                pt[1].x = 3 * cx / 8 ;  pt[1].y = 5 * cy / 8 ;
                pt[2].x = 1 * cx / 8 ;  pt[2].y = 4 * cy / 8 ;

                Triangle (pdis -> hDC, pt) ;
                break ;
           }

                // Invert the rectangle if the button is selected

           if (pdis -> itemState & ODS_SELECTED)
                InvertRect (pdis -> hDC, &pdis -> rcItem) ;

                // Draw a focus rectangle if the button has the focus

           if (pdis -> itemState & ODS_FOCUS)
           {
                pdis -> rcItem.left   += cx / 16 ;
                pdis -> rcItem.top    += cy / 16 ;
                pdis -> rcItem.right  -= cx / 16 ;
                pdis -> rcItem.bottom -= cy / 16 ;

                DrawFocusRect (pdis -> hDC, &pdis -> rcItem) ;
           }
           return 0 ;

      case WM_DESTROY :
           PostQuitMessage (0) ;
           return 0 ;
      }
      return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

该程序在其客户区的中央包含了两个按钮，如图 9-2 所示。左边的按钮有 4 个三角形指向按钮的中央，按下该按钮时，窗口的尺寸将缩小 10%。右边的按钮有 4 个向外指的三角形，按下该按钮时，窗口的尺寸将增大 10%。

如果您只需要在按钮中显示图标或位图，您可以用 BS_ICON 或 BS_BITMAP 风格，并用 BM_SETIMAGE 消息设置位图。但是，对于 BS_OWNERDRAW 风格的按钮，它允许

完全自由地绘制按钮。

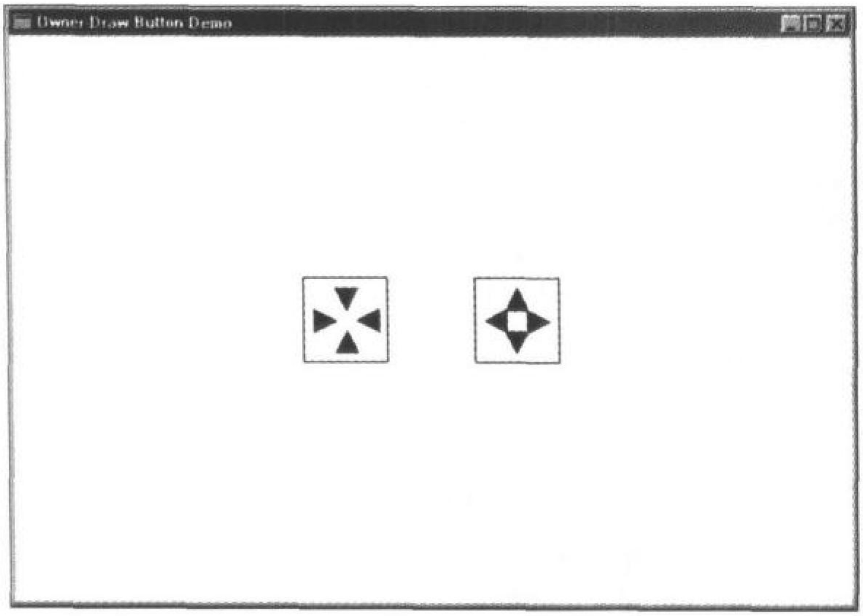


图 9-2　OWNDRAW 的屏幕显示

在处理 WM_CREATE 消息期间,OWNDRAW 建立两个 BS_OWNERDRAW 风格的按钮;按钮的宽度是系统字体的 8 倍,高度是系统字体的 4 倍。(在使用预先定义好的位图绘制按钮时,这些尺寸在 VGA 上建立的按钮为 64 像素宽 64 像素高,知道这些数据非常有用。)这些按钮尚未定位,在处理 WM_SIZE 消息期间,通过调用 MoveWindow 函数,OWNDRAW 将按钮定位在客户区的中心。

按下这些按钮时,它们就会产生 WM_COMMAND 消息。为了处理这些 WM_COMMAND 消息,OWNDRAW 调用 GetWindowRect,将整个窗口(不只是客户区)的位置和尺寸存放在 RECT(矩形)结构中,这个位置相对于屏幕。然后,根据按下的是左边的还是右边的按钮,OWNDRAW 调节这个矩形结构的各个域值。程序再通过调用 MoveWindow 来重新确定位置和尺寸。这将产生另一个 WM_SIZE 消息,按钮被重新定位在客户区的中央。

如果这是程序所做的全部处理,那么这完全可以,只不过按钮是不可见的。使用 BS_OWNERDRAW 风格建立的按钮会在需要重新着色的任何时候都向它的父窗口发送一个 WM_DRAWITEM 消息。这出现在以下几种情况中:当按钮被创建时,当按钮被按下或者被释放时,当按钮得到或者失去输入焦点时,以及当按钮需要重新着色的任何时候。

在处理 WM_DRAWITEM 消息期间,lParam 消息参数是指向类型 DRAWITEM-

STRUCT 结构的指针,OWNDRAW 程序将这个指针保存在 pdis 变量中,这个结构包含了画该按钮时程序所必须的消息(这个结构也可以为自绘列表框和菜单所使用)。对按钮而言非常重要的结构域有 hDC (按钮的设备描述表)、rcItem (提供按钮尺寸的 RECT 结构)、CtlID (控制窗口 ID)和 itemState (它说明按钮是否被按下或者按钮是否拥有输入焦点)。

调用 FillRect 用白色画刷抹掉按钮的表面,调用 FrameRect 在按钮的周围画上黑框,由此 OWNDRAW 便启动了 WM_DRAWITEM 处理过程。然后,通过调用 Polygon,OWNDRAW 在按钮上画出 4 个黑色实心的三角形。这是通常的情形。

如果按钮当前被按下,那么 DRAWITEMSTRUCT 的 itemState 域中的某位将被置位。您可以使用 ODS_SELECTED 常量来测试这些位。如果这些位被置位,那么 OWNDRAW 将通过调用 InvertRect 将按钮翻转为相反的颜色。如果按钮拥有输入焦点,那么 itemState 的 ODS_FOCUS 位将被置位。在这种情况下,OWNDRAW 通过调用 DrawFocusRect,在按钮的边界内画一个虚线的矩形。

在使用拥有者绘制按钮时应该注意以下几个方面:Windows 获得设备描述表并将其作为 DRAWITEMSTRUCT 结构的一个域;保持设备描述表处于您找到它时所处的状态,任何被选进设备描述表的 GDI 对象都必需被释放;另外,当心不要在定义按钮边界的矩形外面进行绘制。

9.3 静态类

在 CreateWindow 函数中指定窗口类为"static",您就可以建立静态的子窗口控制。这些子窗口非常"文静"。它既不接收鼠标或键盘输入,也不向父窗口发送 WM_COMMAND 消息。

当您在静态子窗口上移动或者按下鼠标时,这个子窗口将捕获 WM_NCHITTEST 消息,并将 HTTRANSPARENT 的值返回给 Windows,这将使 Windows 向其下层窗口(通常是它的父窗口)发送相同的 WM_NCHITTEST 消息。父窗口常常将该消息传递给 DefWindowProc,在这里,它被转换为客户区的鼠标消息。

前 6 个静态窗口风格只简单地在子窗口的客户区内画一个矩形或者边框。在下表的上部,"RECT"静态风格(左列)是填充矩形的风格;3 个"FRAME"风格(右列)是没有填充的矩形轮廓:

SS_BLACKRECT	SS_BLACKFRAME
SS_GRAYRECT	SS_GRAYFRAME
SS_WHITERECT	SS_WHITEFRAME

"BLACK"、"GRAY"、"WHITE"并不意味着黑、灰和白色,这些颜色是由系统颜色决

定的，如表 9-4 所示。

表 9-4

静态控制	系统颜色
BLACK	COLOR_3DDKSHADOW
GRAY	COLOR_BTNSHADOW
WHITE	COLOR_BTNHIGHLIGHT

对于这些风格，CreateWindow 调用中的窗口文本域被忽略。矩形的左上角开始于 x 位置坐标和 y 位置坐标，这些坐标都相对于父窗口。您也可以使用 SS_ETCHEDHORZ、SS_ETCHEDVERT 或者 SS_ETCHEDFRAME 风格，采用灰色和白色来创建一个带有阴影的边框。

静态类也包括了三种文本风格：SS_LEFT、SS_RIGHT 和 SS_CENTER。它们创建左对齐、右对齐和居中文本。文本在 CreateWindow 调用的窗口文本参数中给出，并且在以后可以用 SetWindowText 来改变它。当静态控制的窗口过程显示文本时，它使用 DrawText 函数以及 DT_WORDBREAK、DT_NOCLIP 和 DT_EXPANDTABS 参数。文本在子窗口的矩形内可以按字进行换行。

这三种文本风格子窗口的背景通常为 COLOR_BTNFACE，而文本本身是 COLOR_WINDOWTEXT。在捕获 WM_CTLCOLORSTATIC 消息时，您可以通过调用 SetTextColor 来改变文本颜色，通过 SetBkColor 来改变背景颜色，并返回背景画刷句柄。后面的 COLORS1 程序演示了这一点。

最后，静态类还包括了窗口风格 SS_ICON 和 SS_USERITEM，但是当它们被用作子窗口控制时却没有任何意义。我们在讨论对话框时还要涉及它们。

9.4 滚动条类

我在第四章首次讨论了滚动条，也讨论了“窗口滚动条”和“滚动控制”之间的一些区别。SYSMETS 程序使用窗口滚动条，它出现在窗口的右下角。您可以在创建窗口时通过将标识符 WS_VSCROLL、WS_HSCROLL 或者两者都包含在窗口风格中，从而给窗口添加滚动条。现在我们准备建立一些滚动条控制，它们是可以在父窗口的客户区的任何地方出现的子窗口。您可以使用预先定义的窗口类“scrollbar”以及两个滚动条风格 SBS_VERT 和 SBS_HORZ 中的一个来建立子窗口滚动条控制。

与按钮控制（以及将在后面讨论的编辑和列表框控制）不同，滚动条控制不向父窗口发送 WM_COMMAND 消息，而是像窗口滚动条那样发送 WM_VSCROLL 和 WM_HSCROLL 消息。在处理滚动条消息时，您可以通过 lParam 参数来区分开窗口滚动条与

滚动条控制。对于子窗口滚动条其值为 0,对于滚动条控制其值为滚动条窗口句柄。对窗口滚动条和滚动条控制来说,wParam 参数的高位字和低位字的含义相同。

虽然窗口滚动条有固定的宽度,Windows 使用 CreateWindow 调用中(或者在后面的 MoveWindow 调用中)给出的矩形尺寸来确定滚动条控制的尺寸。您可以建立细而长的滚动条控制,也可以建立短而粗的滚动条控制。

如果您想建立与窗口滚动条尺寸相同的滚动条控制,那么可以使用 GetSystemMetrics 获取水平滚动条的高度:

```
GetSystemMetrics (SM_CYHSCROLL) ;
```

或者垂直滚动条的宽度:

```
GetSystemMetrics (SM_CXVSCROLL) ;
```

根据 Windows 文档,滚动条窗口风格标识符 SBS_LEFTALIGN、SBS_RIGHTALIGN、SBS_TOP ALIGN 和 SBS_BOTTOMALIGN 给出滚动条的标准尺寸,但是这些风格只在对话框中对滚动条有效。

对于窗口滚动条,您可以使用同样的调用来设置滚动条控制的范围和位置:

```
SetScrollRange (hwndScroll, SB_CTL, iMin, iMax, bRedraw) ;
SetScrollPos (hwndScroll, SB_CTL, iPos, bRedraw) ;
SetScrollInfo (hwndScroll, SB_CTL, &si, bRedraw) ;
```

其区别在于:窗口滚动条将父窗口的句柄作为第一个参数,并且 SB_VERT 或者 SB_HORZ 作为第二个参数。

令人吃惊的是,名为 COLOR_SCROLLBAR 的系统颜色不再用于滚动条。两端的按钮和小方块的颜色由 COLOR_BTNFACE、COLOR_BTNHILIGHT、COLOR_BTNSHADOW、COLOR_BTNTEXT (用于小箭头)、COLOR_DKSHADOW,以及 COLOR_BTNLIGHT 决定。两端按钮之间较大的区域颜色是由 COLOR_BTNFACE 和 COLOR_BTNHIGHLIGHT 一起来确定的。

如果您捕获了 WM_CTLCOLORSCROLLBAR 消息,那么可以在消息处理中返回画刷以取代该颜色。让我们试一下。

9.4.1 COLORS1 程序

为了解滚动条和静态子窗口的一些用法——也为了深入了解颜色——我们将使用 COLORS1 程序,如程序 9-3 所示。COLORS1 在客户区的左半部显示三种滚动条,并分别标以“Red”、“ Green”和“Blue”。当您滚动滚动条时,客户区的右半部将变为三种原色混合而成的合成色,三种原色的数值显示在三个滚动条的下面。

程序 9-3 COLORS1 程序

COLORS1.C

```
/*------------------------------------------------
   COLORS1.C—Colors Using Scroll Bars
             (c) Charles Petzold, 1998
  ------------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc    (HWND, UINT, WPARAM, LPARAM) ;
LRESULT CALLBACK ScrollProc (HWND, UINT, WPARAM, LPARAM) ;

int     idFocus ;
WNDPROC OldScroll[3] ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("Colors1") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = CreateSolidBrush (0) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }
     hwnd = CreateWindow (szAppName, TEXT ("Color Scroll"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;
```

```
    ShowWindow (hwnd, iCmdShow) ;
    UpdateWindow (hwnd) ;

    while (GetMessage (&msg, NULL, 0, 0))
    {
         TranslateMessage (&msg) ;
         DispatchMessage  (&msg) ;
    }
    return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static COLORREF crPrim[3] = { RGB (255, 0, 0), RGB (0, 255, 0),
                                  RGB (0, 0, 255) } ;
    static HBRUSH   hBrush[3], hBrushStatic ;
    static HWND     hwndScroll[3], hwndLabel[3], hwndValue[3], hwndRect ;
    static int      color[3], cyChar ;
    static RECT     rcColor ;
    static TCHAR *  szColorLabel[] = { TEXT ("Red"), TEXT ("Green"),
                                       TEXT ("Blue") } ;
    HINSTANCE       hInstance ;
    int             i, cxClient, cyClient ;
    TCHAR           szBuffer[10] ;

    switch (message)
    {
    case WM_CREATE :
        hInstance = (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE) ;

             // Create the white-rectangle window against which the
             // scroll bars will be positioned. The child window ID is 9.

        hwndRect = CreateWindow (TEXT ("static"), NULL,
                                 WS_CHILD | WS_VISIBLE | SS_WHITERECT,
                                 0, 0, 0, 0,
                                 hwnd, (HMENU) 9, hInstance, NULL) ;

        for (i = 0 ; i < 3 ; i++)
        {
                  // The three scroll bars have IDs 0, 1, and 2, with
                  // scroll bar ranges from 0 through 255.

             hwndScroll[i] = CreateWindow (TEXT ("scrollbar"), NULL,
                                           WS_CHILD | WS_VISIBLE |
                                           WS_TABSTOP | SBS_VERT,
                                           0, 0, 0, 0,
```

```
                                hwnd, (HMENU) i, hInstance, NULL) ;

          SetScrollRange (hwndScroll[i], SB_CTL, 0, 255, FALSE) ;
          SetScrollPos   (hwndScroll[i], SB_CTL, 0, FALSE) ;

               // The three color-name labels have IDs 3, 4, and 5,
               // and text strings "Red", "Green", and "Blue".

          hwndLabel [i] = CreateWindow (TEXT ("static"), szColorLabel[i],
                                 WS_CHILD | WS_VISIBLE | SS_CENTER,
                                 0, 0, 0, 0,
                                 hwnd, (HMENU) (i + 3),
                                 hInstance, NULL) ;

               // The three color-value text fields have IDs 6, 7,
               // and 8, and initial text strings of "0".

          hwndValue [i] = CreateWindow (TEXT ("static"), TEXT ("0"),
                                 WS_CHILD | WS_VISIBLE | SS_CENTER,
                                 0, 0, 0, 0,
                                 hwnd, (HMENU) (i + 6),
                                 hInstance, NULL) ;

          OldScroll[i] = (WNDPROC) SetWindowLong (hwndScroll[i],
                                 GWL_WNDPROC, (LONG) ScrollProc) ;

          hBrush[i] = CreateSolidBrush (crPrim[i]) ;
     }

     hBrushStatic = CreateSolidBrush (
                         GetSysColor (COLOR_BTNHIGHLIGHT)) ;

     cyChar = HIWORD (GetDialogBaseUnits ()) ;
     return 0 ;

case WM_SIZE :
     cxClient = LOWORD (lParam) ;
     cyClient = HIWORD (lParam) ;
     SetRect (&rcColor, cxClient / 2, 0, cxClient, cyClient) ;

     MoveWindow (hwndRect, 0, 0, cxClient / 2, cyClient, TRUE) ;

     for (i = 0 ; i < 3 ; i++)
     {
          MoveWindow (hwndScroll[i],
                      (2 * i + 1) * cxClient / 14, 2 * cyChar,
                      cxClient / 14, cyClient - 4 * cyChar, TRUE) ;
```

```
          MoveWindow (hwndLabel[i],
                      (4 * i + 1) * cxClient / 28, cyChar / 2,
                      cxClient / 7, cyChar, TRUE) ;

          MoveWindow (hwndValue[i],
                      (4 * i + 1) * cxClient / 28,
                      cyClient - 3 * cyChar / 2,
                      cxClient / 7, cyChar, TRUE) ;
     }
     SetFocus (hwnd) ;
     return 0 ;

case WM_SETFOCUS :
     SetFocus (hwndScroll[idFocus]) ;
     return 0 ;

case WM_VSCROLL :
     i = GetWindowLong ((HWND) lParam, GWL_ID) ;

     switch (LOWORD (wParam))
     {
     case SB_PAGEDOWN :
          color[i] += 15 ;
                                   // fall through
     case SB_LINEDOWN :
          color[i] = min (255, color[i] + 1) ;
          break ;

     case SB_PAGEUP :
          color[i] -= 15 ;
                                   // fall through
     case SB_LINEUP :
          color[i] = max (0, color[i] - 1) ;
          break ;

     case SB_TOP :
          color[i] = 0 ;
          break ;

     case SB_BOTTOM :
          color[i] = 255 ;
          break ;

     case SB_THUMBPOSITION :
     case SB_THUMBTRACK :
          color[i] = HIWORD (wParam) ;
          break ;
```

```
          default :
               break ;
          }
          SetScrollPos (hwndScroll[i], SB_CTL, color[i], TRUE) ;
          wsprintf (szBuffer, TEXT ("%i"), color[i]) ;
          SetWindowText (hwndValue[i], szBuffer) ;

          DeleteObject ((HBRUSH)
               SetClassLong (hwnd, GCL_HBRBACKGROUND, (LONG)
                    CreateSolidBrush (RGB (color[0], color[1], color[2])))) ;

          InvalidateRect (hwnd, &rcColor, TRUE) ;
          return 0 ;

     case WM_CTLCOLORSCROLLBAR :
          i = GetWindowLong ((HWND) lParam, GWL_ID) ;
          return (LRESULT) hBrush[i] ;

     case WM_CTLCOLORSTATIC :
          i = GetWindowLong ((HWND) lParam, GWL_ID) ;

          if (i >= 3 && i <= 8)         // static text controls
          {
               SetTextColor ((HDC) wParam, crPrim[i % 3]) ;
               SetBkColor ((HDC) wParam, GetSysColor (COLOR_BTNHIGHLIGHT));
               return (LRESULT) hBrushStatic ;
          }
          break ;

     case WM_SYSCOLORCHANGE :
          DeleteObject (hBrushStatic) ;
          hBrushStatic = CreateSolidBrush (GetSysColor (COLOR_BTNHIGHLIGHT)) ;
          return 0 ;
     case WM_DESTROY :
          DeleteObject ((HBRUSH)
               SetClassLong (hwnd, GCL_HBRBACKGROUND, (LONG)
                    GetStockObject (WHITE_BRUSH))) ;

          for (i = 0 ; i < 3 ; i++)
               DeleteObject (hBrush[i]) ;

          DeleteObject (hBrushStatic) ;
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
```

```
}

LRESULT CALLBACK ScrollProc (HWND hwnd, UINT message,
                             WPARAM wParam, LPARAM lParam)
{
     int id = GetWindowLong (hwnd, GWL_ID) ;

     switch (message)
     {
     case WM_KEYDOWN :
          if (wParam == VK_TAB)
               SetFocus (GetDlgItem (GetParent (hwnd),
                    (id + (GetKeyState (VK_SHIFT) < 0 ? 2 : 1)) % 3)) ;
          break ;

     case WM_SETFOCUS :
          idFocus = id ;
          break ;
     }
     return CallWindowProc (OldScroll[id], hwnd, message, wParam, lParam) ;
}
```

COLORS1 利用子窗口进行工作，该程序使用 10 个子窗口控制：3 个滚动条、6 个静态文本窗口和 1 个静态矩形框。COLORS1 捕获 WM_CTLCOLORSCROLLBAR 消息来给红、绿和蓝色 3 个滚动条的内部着色，并捕获 WM_CTLCOLORSTATIC 消息来着色静态文本。

您可以使用鼠标或者键盘来滚动滚动条，从而利用 COLORS1 作为一种开发工具尝试颜色的使用，为您自己的 Windows 程序选择漂亮的颜色（或者，您可能更喜欢难看的颜色）。COLORS1 的显示如下页图 9-3 所示。不幸的是，这些颜色在打印纸上被显示为不同深度的灰色。

COLORS1 不处理 WM_PAINT 消息，它的几乎所有工作都是由子窗口完成的。

客户区右半部显示的颜色实际上是窗口的背景颜色。SS_WHITERECT 风格的静态子窗口显示在客户区的左半部。3 个滚动条是 SBS_VERT 风格的子窗口控制，它们被定位在 SS_WHITERECT 子窗口的顶部。另外 6 个 SS_CENTER 风格（居中文本）的静态子窗口提供标签和颜色值。COLORS1 在 WinMain 函数中用 CreateWindow 创建它的普通的重叠式窗口和 10 个子窗口。SS_WHITERECT 和 SS_CENTER 静态窗口使用窗口类“static”；3 个滚动条使用窗口类“scrollbar”。

CreateWindow 调用中的 x 位置、y 位置、宽度和高度参数最初置为 0，因为位置和大小都取决于客户区的尺寸，而它目前尚未确定。COLORS1 的窗口过程在接收到 WM_SIZE 消息时，就使用 MoveWindow 给 10 个子窗口重新确定大小。所以，每当您对 COL-

ORS1 窗口进行缩放时，滚动条的尺寸就会按比例变化。

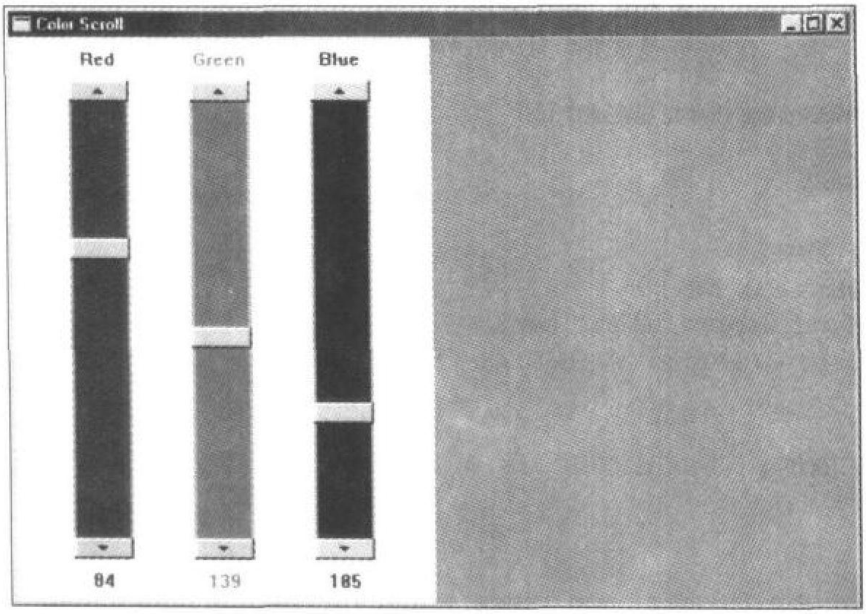


图 9-3　COLORS1 的屏幕显示

当 WndProc 窗口过程收到 WM_VSCROLL 消息时，lParam 参数的高位字就是子窗口的句柄。我们可以使用 GetWindowWord 来得到子窗口的 ID 号：

```
i = GetWindowLong ((HWND) lParam, GWL_ID) ;
```

对于这三个滚动条，我们已经按习惯将其 ID 置为 0、1、2，所以 WndProc 能区别出是哪个滚动条在产生消息。

由于子窗口的句柄在创建时就被保存在数组中，所以 WndProc 就能对相应滚动条消息进行处理，并通过调用 SetScrollPos 来设置相应的新值：

```
SetScrollPos (hwndScroll[i], SB_CTL, color[i], TRUE) ;
```

WndProc 也改变滚动条底部的子窗口的文本：

```
wsprintf (szBuffer, TEXT ("%i"), color[I]) ;
SetWindowText (hwndValue[i], szBuffer) ;
```

9.4.2 自动键盘接口

滚动条控制也能处理键盘输入，但是只能在拥有输入焦点时。表 9-5 说明怎样将键盘光标键转变为滚动条消息。

事实上，SB_TOP 和 SB_BOTTOM 滚动条消息只能用键盘产生。在使用鼠标点击滚

动条时，如果想使该滚动条获得输入焦点，那么您必须将 WS_TABSTOP 标识符包含到 CreateWindow 调用的窗口类参数中。当滚动条拥有输入焦点时，在该滚动条的小方框上将显示一个闪烁的灰色块。

表 9-5

光标键	滚动条消息的 wParam 值
Home	SB_TOP
End	SB_BOTTOM
Page Up	SB_PAGEUP
Page Down	SB_PAGEDOWN
左或上	SB_LINEUP
右或下	SB_LINEDOWN

为了给滚动条提供全面的键盘接口，还需要另外一些工作。首先，WndProc 窗口过程必须使滚动条拥有输入焦点，它是通过处理 WM_SETFOCUS 消息来完成这一点的，该 WM_SETFOCUS 消息是当滚动条获得输入焦点时其父窗口接收到的。WndProc 给其中一个滚动条设置输入焦点。

```
SetFocus (hwndScroll[idFocus]) ;
```

其中 idFocus 是一个全局变量。

但是，还需要一些借助键盘尤其是 Tab 键来从一个滚动条转换到另一个滚动条的方法。这比较困难，因为一旦某个滚动条拥有了输入焦点，它就处理所有的键盘输入，但滚动条只关心光标键，而忽略 Tab 键。解决这一两难处境的方法是“窗口子类化”。我们将用它来给 COLORS1 增加使用 Tab 键从一个滚动条跳到另一个滚动条的功能。

9.4.3 窗口子类化

滚动条控制的窗口过程是 Windows 内部的。但是，将 GWL_WNDPROC 标识符作为参数来调用 GetWindowLong，您就可以得到这个窗口过程的地址。另外，您可以调用 SetWindowLong 给该滚动条设置一个新的窗口过程，这个技术叫做“窗口子类化”，非常有用。它能让您给现存的窗口过程设置“钩子”，以便在自己的程序中处理一些消息，同时将其他所有消息传递给旧的窗口过程。

在 COLORS1 中对滚动条消息进行预处理的窗口过程叫做 ScrollProc，它在 COLORS1.C 文件的尾部。由于 ScrollProc 是 COLORS1 中的函数，而 Windows 将调用 COLORS1，所以 ScrollProc 必须被定义为回调函数。

对三个滚动条中的每一个，COLORS1 使用 SetWindowLong 来设置新的滚动条窗口过程的地址并获取现存滚动条窗口过程的地址：

```
OldScroll[i] = (WNDPROC) SetWindowLong (hwndScroll[i], GWL_WNDPROC,
                                        (LONG) ScrollProc)) ;
```

现在，函数 ScrollProc 得到了 Windows 发送到 COLORS1 中三个滚动条（当然不是其他程序中的滚动条）的滚动条窗口过程的全部消息。ScrollProc 窗口过程在接收到 Tab 或者 Shift - Tab 键时，就将输入焦点改变到下一个（或者上一个）滚动条。它使用 CallWindowProc 调用旧的滚动条窗口过程。

9.4.4 给背景着色

当 COLORS1 定义它的窗口类时，也为其客户区背景定义了一个实心的黑色画刷：

```
wndclass.hbrBackground = CreateSolidBrush (0) ;
```

当您改变 COLORS1 的滚动条设置时，程序必须建立一个新的画刷，并将该新画刷句柄放入窗口类结构中。如同使用 GetWindowLong 和 SetWindowLong 能得到并设置滚动条窗口过程一样，用 GetClassWord 和 SetClassWord 能得到这个画刷的句柄。

您可以建立新的画刷并将其句柄插入窗口类结构中，然后删除旧的画刷：

```
DeleteObject ((HBRUSH)
    SetClassLong (hwnd, GCL_HBRBACKGROUND, (LONG)
        CreateSolidBrush (RGB (color[0], color[1], color[2])))) ;
```

Windows 下一次重新给窗口的背景着色时，将使用这个新画刷。为了强迫 Windows 抹掉背景，我们将使整个客户区无效：

```
InvalidateRect (hwnd, &rcColor, TRUE) ;
```

TRUE（非零）值作为第三个参数表示希望在重新着色之前删去背景。

InvalidateRect 使 Windows 在窗口过程的消息队列中放进一个 WM_PAINT 消息。由于 WM_PAINT 消息的级别比较低，所以，如果您还在使用鼠标或者光标键移动滚动条的话，这条消息将不会立即被处理。如果您想在颜色改变之后使该窗口立即变成最新的（当前的），那么您可以在 InvalidateRect 之后增加语句

```
UpdateWindow (hwnd) ;
```

但是这会使得键盘和鼠标处理速度变慢。

COLORS1 中的 WndProc 函数不处理 WM_PAINT 消息，而是将其传给 DefWindowProc。Windows 对 WM_PAINT 消息的默认处理只是调用 BeginPaint 和 EndPaint 使窗口生效。因为在 InvalidateRect 调用中已经指定背景要被抹掉，所以 BeginPaint 调用使 Windows 发出一个 WM_ERASEBKGND（删除背景）消息，WndProc 也将忽略这个消息。Windows 用窗口类中指定的画刷将客户区的背景抹去，这样就处理了这个消息。

在终止以前进行清除总是一个好主意，因此在处理 WM_DESTROY 消息期间，再一次调用 DeleteObject：

```
DeleteObject ((HBRUSH)
   SetClassLong (hwnd, GCL_HBRBACKGROUND,
      (LONG) GetStockObject (WHITE_BRUSH))) ;
```

9.4.5 给滚动条和静态文本着色

在 COLORS1 中，3 个滚动条的内部和 6 个文本域中的文本着色为红、绿和蓝色。滚动条的着色是通过处理 WM_CTLCOLORSCROLLBAR 消息来完成的。

在 WndProc 中，我们为画刷定义了一个由三个句柄组成的静态数组：

```
static HBRUSH hBrush [3] ;
```

在处理 WM_CREATE 期间，我们建立三个画刷：

```
for (I = 0 ; I < 3 ; I++)
   hBrush[0] = CreateSolidBrush (crPrim [I]) ;
```

其中 crPrim 数组中包含三种原色的 RGB 值。在 WM_CTLCOLORSCROLLBAR 处理期间窗口过程返回这三个画刷中的一个：

```
case WM_CTLCOLORSCROLLBAR:
   i = GetWindowLong ((HWND) lParam, GWL_ID) ;
   return (LRESULT) hBrush [i] ;
```

在处理 WM_DESTROY 消息的过程中，这些画刷必须被删除：

```
for (i = 0 ; i < 3 ; i++)
   DeleteObject (hBrush [i]) ;
```

同样，静态文本域中的文本是在处理 WM_CTLCOLORSTATIC 消息中调用 SetTextColor 来着色的。文本背景用 SetBkColor 函数设置为系统颜色 COLOR_BTNHIGHLIGHT，这导致文本背景颜色和滚动条与文本后面的静态矩形控制的颜色一样。对于静态文本控制，这种文本背景颜色只用于字符串中每个字符后面的矩形，而不会用于整个控制窗口。为了实现这一点，窗口过程还必须返回 COLOR_BTNHIGHLIGHT 颜色画刷的句柄。这个画刷被称为 hBrushStatic，它在 WM_CREATE 消息期间创建，在 WM_DESTROY 消息期间清除。

伴随 WM_CREATE 消息期间基于 COLOR_BTNHIGHLIGHT 颜色创建画刷，并且在该程序运行期间使用这一画刷，我们遇到了一个小问题。如果程序在运行期间改变了 COLOR_BTNHIGHLIGHT 颜色，那么静态矩形的颜色将发生变化，并且文本背景的颜色

也会变化，但是文本窗口控制的整个背景将保持原有的 COLOR_BTNHIGHLIGHT 颜色。

为了解决这个问题，COLORS1 也简单地通过使用新颜色重新创建 hBrushStatic 来处理 WM_SYSCOLORCHANGE 消息。

9.5 编辑类

在某些方面，编辑类是最简单的预定义窗口类；在另一方面，它又是最复杂的窗口类。当您使用类名"edit"建立子窗口时，根据 CreateWindow 调用中的 x 位置、y 位置、宽度和高度等这些参数定义了一个矩形。此矩形含有可编辑文本。当子窗口控制拥有输入焦点时，您可以输入文本，移动光标，使用鼠标或者 Shift 键与一个光标键来选取部分文本，按 Ctrl-X 来删除所选文本或按 Ctrl-C 来复制所选文本并送到剪贴板中，按 Ctrl-V 键插入剪贴板中的文本。

编辑控制的最简单应用之一是作为单行输入区域。但是编辑控制并不仅限于单行，这一点我将在程序 9-4 中说明。和我们在这本书中所遇到的各种其他问题一样，POPPAD 程序将逐步增强以使用菜单、对话框(加载和保存文件)和打印。最后的版本将是一个简单而完整的文本编辑器，并且其代码将非常简洁。

程序 9-4　POPPAD1 程序

POPPAD1.C

```
/*---------------------------------------------------------
     POPPAD1.C——Popup Editor using child window edit box
                 (c) Charles Petzold, 1998
  ---------------------------------------------------------*/

#include <windows.h>

#define ID_EDIT     1

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

TCHAR szAppName[] = TEXT ("PopPad1") ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style          = CS_HREDRAW | CS_VREDRAW ;
```

```
     wndclass.lpfnWndProc    = WndProc ;
     wndclass.cbClsExtra     = 0 ;
     wndclass.cbWndExtra     = 0 ;
     wndclass.hInstance      = hInstance ;
     wndclass.hIcon          = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground  = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName   = NULL ;
     wndclass.lpszClassName  = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, szAppName,
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;
     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static HWND hwndEdit ;

     switch (message)
     {
     case WM_CREATE :
          hwndEdit = CreateWindow (TEXT ("edit"), NULL,
                         WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL |
                                   WS_BORDER | ES_LEFT | ES_MULTILINE |
                                   ES_AUTOHSCROLL | ES_AUTOVSCROLL,
                         0, 0, 0, 0, hwnd, (HMENU) ID_EDIT,
                         ((LPCREATESTRUCT) lParam) -> hInstance, NULL) ;
          return 0 ;
```

```
          case WM_SETFOCUS :
               SetFocus (hwndEdit) ;
               return 0 ;

          case WM_SIZE :
               MoveWindow (hwndEdit, 0, 0, LOWORD (lParam), HIWORD (lParam), TRUE) ;
               return 0 ;

          case WM_COMMAND :
               if (LOWORD (wParam) == ID_EDIT)
                    if (HIWORD (wParam) == EN_ERRSPACE ||
                              HIWORD (wParam) == EN_MAXTEXT)
                         MessageBox (hwnd, TEXT ("Edit control out of space."),
                                     szAppName, MB_OK | MB_ICONSTOP) ;
               return 0 ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
          return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

POPPAD1 是一个多行编辑器(只是没有文件 I/O),其 C 语言源代码不到 100 行。(不过,有一个缺陷,即预定义的多行编辑控制只限于 30 000 字符的文本。)您可以看到,POPPAD1 本身并没有做多少工作,预定义的编辑控制完成了许多工作,这样,您可以知道无需额外的程序时编辑控制能做些什么。

9.5.1 编辑类风格

如前面所提到的,在 CreateWindow 调用中将“edit”作为窗口类建立了一个编辑控制,窗口风格是 WS_CHILD 加上几个选项。如同在静态子窗口控制中一样,编辑控制中的文本可以左对齐、右对齐或者居中,使用窗口风格 ES_LEFT、ES_RIGHT 和 ES_CENTER 来指定这种格式。

默认时,编辑控制是单行的。使用 ES_MULTILINE 窗口风格可以建立多行编辑控制。对于单行编辑控制,一般只可以在编辑控制矩形的尾部输入正文。要建立一个自动水平滚动的编辑控制,您可以采用风格 ES_AUTOHSCROLL。对一个多行编辑控制,正文不会整字回行,除非使用 ES_AUTOHSCROLL 风格。在这种情况下,您必须按 Enter 键来开始新的一行。您还可以使用风格 ES_AUTOVSCROLL 来将垂直滚动条包括在多行编辑控制中。

当您在多行编辑控制中包括这些滚动风格时，也许还想给编辑控制增加滚动条。要做到这些，可以对非子窗口使用同一窗口风格标识符 WS_HSCROLL 和 WS_VSCROLL。默认时，编辑控制没有边界，使用风格 WS_BORDER 可以增加边界。

当您在编辑控制中选择文本时，Windows 将选择的文本反显。但是当编辑控制失去输入焦点时，选定的文本将不再被加亮。如果希望在编辑控制没有输入焦点时选定的文本仍然被加亮，您可以使用风格 ES_NOHIDESEL。

在 POPPAD1 创建其编辑控制时，CreateWindow 调用按如下形式给出风格：

```
WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL |
     WS_BORDER | ES_LEFT | ES_MULTILINE |
     ES_AUTOHSCROLL | ES_AUTOVSCROLL
```

在 POPPAD1 中，编辑控制的大小是后来当 WndProc 接收到 WM_SIZE 消息时通过调用 MoveWindow 来定义的。编辑控制的尺寸被简单地设置为主窗口的尺寸：

```
MoveWindow (hwndEdit, 0, 0, LOWORD (lParam),
                    HIWORD (lParam), TRUE) ;
```

对于单行编辑控制，控制的高度必须可以容纳一个字符。如果编辑控制有边界（大多数都有），那么使用一个字符高度的 1.5 倍（包括外部间距）。

9.5.2 编辑控制通知

编辑控制给父窗口过程发送 WM_COMMAND 消息，对按钮控制来说，wParam 和 lParam 变量的含义是相同的：

LOWORD (wParam)	子窗口 ID
HIWORD (wParam)	通知码
lParam	子窗口句柄

通知码如下所示：

EN_SETFOCUS	编辑控制已经获得输入焦点
EN_KILLFOCUS	编辑控制已经失去输入焦点
EN_CHANGE	编辑控制的内容将改变
EN_UPDATE	编辑控制的内容已经改变
EN_ERRSPACE	编辑控制运行已经超出中间
EN_MAXTEXT	编辑控制在插入时运行超出中间
EN_HSCROLL	编辑控制的水平滚动条已经被单击

EN_VSCROLL	编辑控制的垂直滚动条已经被单击

POPPAD1 只捕获 EN_ERRSPACE 和 EN_MAXTEXT 通知码,并显示一个消息框。

9.5.3 使用编辑控制

如果在您的主窗口表面使用了几个单行编辑控制,那么您需要将窗口子类化,以便把输入焦点从一个控制转移到另一个控制。您可以通过捕获 Tab 键和 Shift-Tab 键来完成这种移动,非常像 COLORS1 中所做的(窗口子类化的另一个例子在后面的 HEAD 程序中说明)。如何处理 Enter 键取决于您,可以像 Tab 键那样使用,也可以用作给程序的信号,表示所有的编辑区都准备好了。

如果您想在编辑区中插入文本,那么可以使用 SetWindowText 来做到。将文本从编辑控制中取出涉及到 GetWindowTextLength 和 GetWindowText,我们将在 POPPAD 程序的修订版本中看到这些操作的实例。

9.5.4 发送给编辑控制的消息

因为用 SendMessage 发送给编辑控制的消息很多,并且其中的几个还将在后面 POPPAD 修订版本中用到,所以这里不涉及所有用 SendMessage 发送给编辑控制的消息,只概要地说明一下。

这些消息允许您剪切、复制或者清除当前选择。用户使用鼠标或者 Shift 键加上光标控制键来选择文本并进行上面的操作,这样,在编辑控制中选定的文本将被加亮:

```
SendMessage (hwndEdit, WM_CUT, 0, 0) ;
SendMessage (hwndEdit, WM_COPY, 0, 0) ;
SendMessage (hwndEdit, WM_CLEAR, 0, 0) ;
```

WM_CUT 将当前选择从编辑控制中移走,并将其发送到剪贴板中;WM_COPY 将选定的文本复制到剪贴板中,并保持编辑控制中的内容完好无损;WM_CLEAR 将选定的内容从编辑控制中删除,但是不向剪贴板中发送。

您也可以将剪贴板中的文本插入到编辑控制中的光标位置:

```
SendMessage (hwndEdit, WM_PASTE, 0, 0) ;
```

您可以获取当前选择的起始位置和末尾位置:

```
SendMessage (hwndEdit, EM_GETSEL, (WPARAM) &iStart,
                                  (LPARAM) &iEnd) ;
```

结束位置实际上是最后一个选择字符的位置加 1。

您可以选择文本:

```
SendMessage (hwndEdit, EM_SETSEL, iStart, iEnd) ;
```

您还可以使用其他文本来置换当前的选择：

```
SendMessage (hwndEdit, EM_REPLACESEL, 0, (LPARAM) szString) ;
```

对多行编辑控制，您可以获取行数：

```
iCount = SendMessage (hwndEdit, EM_GETLINECOUNT, 0, 0) ;
```

对任何特定的行，您可以获取距离编辑缓冲区文本开头的偏移量：

```
iOffset = SendMessage (hwndEdit, EM_LINEINDEX, iLine, 0) ;
```

行数从 0 开始计算，iLine 值为 - 1 时返回包含光标所在行的偏移量。您可以获取行的长度：

```
iLength = SendMessage (hwndEdit, EM_LINELENGTH, iLine, 0) ;
```

并将行本身复制到一个缓冲区中：

```
iLength = SendMessage (hwndEdit, EM_GETLINE, iLine, (LPARAM) szBuffer) ;
```

9.6 列表框类

在本章讨论的最后一个预先定义的子窗口控制是列表框。列表框是文本串的集合，这些文本串是一个矩形中可以滚动显示的列状列表。一程序通过向列表框窗口过程发送消息，可以在列表中增加或者删除串。当列表框中的某项被选定时，列表框控制就向其父窗口发送 WM_COMMAND 消息，父窗口也就可以确定选定的是哪一项。

列表框可以是单选的，也可以是多选的，后者允许用户从列表框中选择多个项。当列表框拥有输入焦点时，其中的项的周围显示有虚线。在列表框中，光标位置并不指明选定的项。选定的项被加亮显示，并且是反显的。

在单项选择的列表框中，用户按 Spacebar 键就可以选定光标所在位置的项。方向键移动光标和当前选择指示，并且能够滚动列表框的内容。Page Up 和 Page Down 键也能滚动列表框，但它移动的是光标而不是选定指示。按字母键能将光标和选定指示移到以此字母开头的第一个(或下一个)选项。也可以使用鼠标在要选定的项上单击或者双击来选择它。

在多项选择列表框中，Spacebar 键可以切换光标所在位置的项的选择状态(如果该项已经被选定，则取消选定)。如同在单项选择列表框中一样，方向键取消前面选定的项，并且移动光标和选定指示。但是，Ctrl 键和方向键能够在移动光标的同时不移动选择，Shift 键加方向键能扩展一个选择。

在多项选择列表框中，单击或者双击鼠标键能取消以前所有的选择，而选择被点中

的项。但是，如果在鼠标点中某一项的同时也按下 Shift 键，则只能切换该项的选择状态，而不会改变任何其他项的选择状态。

9.6.1 列表框风格

当您使用 CreateWindow 建立列表框子窗口时，您应该将“listbox”作为窗口类，将 WS_CHILD 作为窗口风格。但是，这个默认列表框风格不向其父窗口发送 WM_COMMAND 消息，这样一来，程序必须向列表框询问其中的项的选择状态(借助于发送给列表框控制的消息)。所以，列表框控制通常都包括列表框风格标识符 LBS_NOTIFY，它允许父窗口接收来自列表框的 WM_COMMAND 消息。如果希望列表框控制对列表框中的项进行排序，那么您可以使用另一种常用的风格 LBS_SORT。

默认情况下，列表框是单项选择的。多项选择的列表框相当少。如果想建立一个多项选择列表框，那么您可以使用风格 LBS_MULTIPLESEL。通常，当给滚动框增加新项时，列表框会修改它自身。您可以通过将风格 LBS_NOREDRAW 包含进去来防止这种现象。但是您也许不想使用这种风格，这时可以使用 WM_SETREDRAW 消息来暂时防止列表框控制的重新着色，我将在稍后讨论 WM_SETREDRAW 消息。

默认时，列表框窗口过程只显示列表项，它的周围没有任何边界。您可以使用窗口风格标识符 WS_BORDER 来加上边界。另外，您可以使用窗口风格标识符 WS_VSCROLL 来增加垂直滚动条，以便用鼠标来滚动列表项。

Windows 头文件定义了一个列表框风格，叫做 LBS_STANDARD，它包含了最常用的风格，其定义如下：

```
(LBS_NOTIFY | LBS_SORT | WS_VSCROLL | WS_BORDER)
```

您也可以采用 WS_SIZEBOX 和 WS_CAPTION 标识符，但是这两个标识符允许您重新定义列表框的大小，也允许您在列表框父窗口的客户区中移动列表框。

列表框的宽度应该能容纳最长串的宽度加上滚动条的宽度。您可以使用

```
GetSystemMetrics (SM_CXVSCROLL) ;
```

来获得垂直滚动条的宽度。您用一个字符的高度乘以想要在视口中显示的项数来计算出列表框的高度。

9.6.2 将串放入列表框

创建列表框之后，下一步是将文本串放入其中，您可以通过调用 SendMessage 给列表框窗口过程发消息来做到这一点。文本串通常通过以 0 开始计数的下标数来引用，其中

0 对应于最顶上的项。在下面的例子中，hwndList 是子窗口列表框控制的句柄，而 iIndex 是下标值。在使用 SendMessage 传递文本串的情况下，lParam 参数是指向 null 结尾串的指针。

在大多数例子中，当窗口过程存储的列表框内容超过了可用内存空间时，SendMessage 将返回 LB_ERRSPACE(定义为 -2)。如果是因为其他原因出的错，那么 SendMessage 将返回 LB_ERR(-1)。如果操作成功，那么 SendMessage 将返回 LB_OKAY(0)。您可以通过测试 SendMessage 的非零值来判明这两种错误。

如果采用 LBS_SORT 风格(或者如果在列表框中按照想要呈现的顺序排列字符串)，那么填充列表框最简单的方法是借助 LB_ADDSTRING 消息：

```
SendMessage (hwndList, LB_ADDSTRING, 0, (LPARAM) szString) ;
```

如果没有采用 LBS_SORT，那么可以使用 LB_INSERTSTRING 指定一个下标值，将字符串插入到列表框中：

```
SendMessage (hwndList, LB_INSERTSTRING, iIndex, (LPARAM) szString) ;
```

例如，如果 iIndex 等于 4，那么 szString 将变为下标值为 4 的串——从顶头开始算起的第 5 个串(因为是从 0 开始计数的)，位于这个点后面的所有串都将向后推移。下标值为 -1 时，将串增加在最后。您可以对风格为 LBS_SORT 的列表框使用 LB_INSERTSTRING，但是这个列表框的内容不能被重新排序。(您也可以使用 LB_DIR 消息将串插入到列表框中，这将在本章的最后进行讨论。)

您可以在指定下标值的同时使用 LB_DELETESTRING 参数，这就可以从列表框中删除串：

```
SendMessage (hwndList, LB_DELETESTRING, iIndex, 0) ;
```

您可以使用 LB_RESETCONTENT 清除列表框中的内容：

```
SendMessage (hwndList, LB_RESETCONTENT, 0, 0) ;
```

当在列表框中增加或者删除串时，列表框窗口过程将更新显示。如果您有许多串需要添加或者删除，那么您也许希望暂时阻止这一动作，其方法是关掉控制的重画标志：

```
SendMessage (hwndList, WM_SETREDRAW, FALSE, 0) ;
```

当您完成后，可以再打开重画标志：

```
SendMessage (hwndList, WM_SETREDRAW, TRUE, 0) ;
```

使用 LBS_NOREDRAW 风格建立的列表框开始时其重画标志是关闭的。

9.6.3 选择和获取项

SendMessage 完成了下面所描述的任务之后，通常返回一个值。如果出错，那么这个值将被设置为 LB_ERR(定义为 -1)。

当列表框中放入一些项之后，您可以弄清楚列表框中有多少项：

```
iCount = SendMessage (hwndList, LB_GETCOUNT, 0, 0) ;
```

其他一些调用对单项选择列表框和多项选择列表框是不同的。让我们先来看看单项选择列表框。

通常，您让用户在列表框中选择条目。但是如果您想加亮显示一个默认选择，则可以使用

```
SendMessage (hwndList, LB_SETCURSEL, iIndex, 0) ;
```

将 iParam 设置为 -1，取消所有选择。

您也可以根据项的第一个字母来选择：

```
iIndex = SendMessage (hwndList, LB_SELECTSTRING, iIndex,
                      (LPARAM) szSearchString) ;
```

在 SendMessage 调用中将 iIndex 作为 iParam 参数时，iIndex 是下标，可以根据它搜索其开头字符与 szSearchString 相匹配的项。iIndex 的值等于 -1 时从头开始搜索，SendMessage 返回被选中项的下标。如果没有开头字符与 szSearchString 相匹配的项时，SendMessage 返回 LB_ERR。

当您得到来自列表框的 WM_COMMAND 消息时(或者在任何其他时候)，您可以使用 LB_GETCURSEL 来确定当前选项的下标：

```
iIndex = SendMessage (hwndList, LB_GETCURSEL, 0, 0) ;
```

如果没有项被选中，那么从调用中返回的 iIndex 值为 LB_ERR。

您可以确定列表框中串的长度：

```
iLength = SendMessage (hwndList, LB_GETTEXTLEN, iIndex, 0) ;
```

并可以将某项复制到文本缓冲区中：

```
iLength = SendMessage (hwndList, LB_GETTEXT, iIndex,
                       (LPARAM) szBuffer) ;
```

在这两种情况下，从调用返回的 iLength 值是字符串的长度。对以 NULL 终结的串的长度来说，szBuffer 数组必须足够大。您也许想用 LB_GETTEXTLEN 先分配一些局部内存

来存放串。

对一个多项选择列表框,您不能使用 LB_SETCURSEL、LB_GETCURSEL 或者 LB_SELECTSTRING,但是您可以使用 LB_SETSEL 来设置某特定项的选择状态,而不影响有可能被选择的其他项:

```
SendMessage (hwndList, LB_SETSEL, wParam, iIndex) ;
```

wParam 参数不为 0 时,选定并加亮某一项;wParam 为 0 时,取消选定。如果 wParam 等于 -1,那么将选定所有项或者取消所有被选定的项。您可以使用

```
iSelect = SendMessage (hwndList, LB_GETSEL, iIndex, 0) ;
```

确定某特定项的选择状态。其中,如果由 iIndex 指定的项被选定,iSelect 被置为非 0,否则被置为 0。

9.6.4 接收来自列表框的消息

当用户用鼠标单击列表框时,列表框将接收输入焦点。下面的操作可以使父窗口将输入焦点转交给列表框控制:

```
SetFocus (hwndList) ;
```

当列表框拥有输入焦点时,光标移动键、字母键和 Spacebar 键都可以用来在该列表框中选择某项。

列表框控制向其父窗口发送 WM_COMMAND 消息,对按钮和编辑控制来说,wParam 和 lParam 变量的含义是相同的:

LOWORD (wParam)	子窗口 ID
HIWORD (wParam)	通知码
lParam	子窗口句柄

通知码及其值如下所示:

LBN_ERRSPACE	-2
LBN_SELCHANGE	1
LBN_DBLCLK	2
LBN_SELCANCEL	3
LBN_SETFOCUS	4
LBN_KILLFOCUS	5

只有列表框窗口风格包括 LBS_NOTIFY 时,列表框控制才向父窗口发送 LBN_SEL-

CHANGE 和 LBN_DBLCLK 码。

LBN_ERRSPACE 码表明列表框已经超出运行空间。LBN_SELCHANGE 码表明当前选择已经被改变;这些消息出现在下列情况下:用户在列表框中移动加亮的项时,用户使用 Spacebar 键切换选择状态时或者用鼠标单击某项时。LBN_DBLCLK 码说明某项已经被鼠标双击。(LBN_SELCHANGE 和 LBN_DBLCLK 通知码的值表示鼠标按动的次数。)

根据应用的需要,您也许要使用 LBN_SELCHANGE 或 LBN_DBLCLK,也许二者都要使用。您的程序会收到许多 LBN_SELCHANGE 消息,但是 LBN_DBLCLK 消息只有当用户双击鼠标时才会出现。如果您的程序使用双击,那么您需要提供一个复制 LBN_DBLCLK 的键盘接口。

9.6.5 一个简单的列表框应用程序

既然您知道了如何建立列表框,如何使用文本项填充列表框,如何接收来自列表框的控制,以及如何获取串,现在是到了写一个应用程序的时候了。如程序 9-5 中所示,ENVIRON 程序在客户区中使用列表框来显示当前操作系统环境变量(例如 PATH 和 WINDIR)。当您选择一个变量时,环境串将显示在客户区的顶部。

程序 9-5 ENVIRON 程序

ENVIRON.C

```
/*---------------------------------------
     ENVIRON.C——Environment List Box
                (c) Charles Petzold, 1998
  ---------------------------------------*/

#include <windows.h>
#define ID_LIST     1
#define ID_TEXT     2

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("Environ") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;
```

```
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Environment List Box"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

void FillListBox (HWND hwndList)
{
     int     iLength ;
     TCHAR * pVarBlock, * pVarBeg, * pVarEnd, * pVarName ;

     pVarBlock = GetEnvironmentStrings () ;  // Get pointer to environment block

     while (* pVarBlock)
     {
          if (* pVarBlock != '=')   // Skip variable names beginning with '='
          {
               pVarBeg = pVarBlock ;                // Beginning of variable name
```

```
                    while (*pVarBlock++ != '=') ;        // Scan until '='
                    pVarEnd = pVarBlock - 1 ;            // Points to '=' sign
                    iLength = pVarEnd - pVarBeg ;        // Length of variable name

                         // Allocate memory for the variable name and terminating
                         // zero. Copy the variable name and append a zero.

                    pVarName = calloc (iLength + 1, sizeof (TCHAR)) ;
                    CopyMemory (pVarName, pVarBeg, iLength * sizeof (TCHAR)) ;
                    pVarName[iLength] = '\0' ;

                         // Put the variable name in the list box and free memory.
                    SendMessage (hwndList, LB_ADDSTRING, 0, (LPARAM) pVarName) ;
                    free (pVarName) ;
               }
               while (*pVarBlock++ != '\0') ; // Scan until terminating zero
          }
          FreeEnvironmentStrings (pVarBlock) ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static HWND  hwndList, hwndText ;
     int          iIndex, iLength, cxChar, cyChar ;
     TCHAR      * pVarName, * pVarValue ;

     switch (message)
     {
     case WM_CREATE :
          cxChar = LOWORD (GetDialogBaseUnits ()) ;
          cyChar = HIWORD (GetDialogBaseUnits ()) ;
               // Create listbox and static text windows.

          hwndList = CreateWindow (TEXT ("listbox"), NULL,
                              WS_CHILD | WS_VISIBLE | LBS_STANDARD,
                              cxChar, cyChar * 3,
                              cxChar * 16 + GetSystemMetrics (SM_CXVSCROLL),
                              cyChar * 5,
                              hwnd, (HMENU) ID_LIST,
                              (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE),
                              NULL) ;

          hwndText = CreateWindow (TEXT ("static"), NULL,
                              WS_CHILD | WS_VISIBLE | SS_LEFT,
                              cxChar, cyChar,
                              GetSystemMetrics (SM_CXSCREEN), cyChar,
                              hwnd, (HMENU) ID_TEXT,
```

```
                    (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE),
                    NULL) ;

          FillListBox (hwndList) ;
          return 0 ;

     case WM_SETFOCUS :
          SetFocus (hwndList) ;
          return 0 ;
     case WM_COMMAND :
          if (LOWORD (wParam) == ID_LIST && HIWORD (wParam) == LBN_SELCHANGE)
          {
                    // Get current selection.

               iIndex = SendMessage (hwndList, LB_GETCURSEL, 0, 0) ;
               iLength = SendMessage (hwndList, LB_GETTEXTLEN, iIndex, 0) + 1 ;
               pVarName = calloc (iLength, sizeof (TCHAR)) ;
               SendMessage (hwndList, LB_GETTEXT, iIndex, (LPARAM) pVarName) ;

                    // Get environment string.

               iLength = GetEnvironmentVariable (pVarName, NULL, 0) ;
               pVarValue = calloc (iLength, sizeof (TCHAR)) ;
               GetEnvironmentVariable (pVarName, pVarValue, iLength) ;

                    // Show it in window.

               SetWindowText (hwndText, pVarValue) ;
               free (pVarName) ;
               free (pVarValue) ;
          }
          return 0 ;

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

ENVIRON 创建两个子窗口:一个是 LBS_STANDARD 风格的列表框,另一个是 SS_LEFT 风格(左对齐文本)的静态窗口。ENVIRON 使用函数 GetEnvironmentStrings 来获得一个指针,该指针指向存有全部环境变量名及其值的内存块。ENVIRON 用 FillListBox 函数来分析此内存块,并使用 LB_ADDSTRING 消息来指定列表框窗口过程将每个串放入列表框中。

当您运行 ENVIRON 时,可以使用鼠标或者键盘来选择环境变量。每次改变选择时,列表框都给其父窗口 WndProc 发送一个 WM_COMMAND 消息。当 WndProc 收到 WM_COMMAND 消息时,它就检查 wParam 的低位字是否为 ID_LIST(列表框的子窗口 ID)和 wParam 的高位字(通知码)是否等于 LBN_SELCHANGE。如果是的,那么它就使用 LB_GETCURSEL 消息来获得选定项目的下标,并使用 LB_GETTEXT 来获得外部环境变量名的文本串本身。ENVIRON 程序使用 C 语言函数 GetEnvironmentVariable 来获得与变量相对应的环境串,使用 SetWindowText 将该串传递到静态子窗口控制中,这个静态子窗口控制被用来显示文本。

9.6.6 文件列表

将最好的留在最后:LB_DIR 是功能最强的列表框消息。它用文件目录表填充列表框,并且可以选择将子目录和有效的磁盘驱动器也包括进来:

```
SendMessage (hwndList, LB_DIR, iAttr, (LPARAM) szFileSpec) ;
```

使用文件属性码

iAttr 参数是文件属性代码,其最低字节可以是表 9-6 中数据组合的文件属性代码。

表 9-6

iAttr	值	属性
DDL_READWRITE	0x0000	普通文件
DDL_READONLY	0x0001	只读文件
DDL_HIDDEN	0x0002	隐含文件
DDL_SYSTEM	0x0004	系统文件
DDL_DIRECTORY	0x0010	子目录
DDL_ARCHIVE	0x0020	有归档位集的文件

高字节提供了一些对所要求的项的附加控制(见表 9-7),前缀 DDL 表示"对话目录列表"。

表 9-7

iAttr	值	选择
DDL_DRIVES	0x4000	包括驱动器的字母
DDL_EXCLUSIVE	0x8000	互斥搜索

当 LB_DIR 消息的 iAttr 值为 DDL_READWRITE 时,列表框列出普通文件、只读文件和具有档案字位集标志的文件。当值为 DDL_DIRECTORY 时,列表框除列出上述文

件之外，还列出子目录，目录位于方括号之内。当值为 DDL_DRIVES | DDL_DIRECTORY 时，那么列表将扩展到包括所有有效的驱动器，驱动器字母显示在虚线之间。

将 iAttr 的最高位置位就可以列出带标志的文件，而不包括普通文件。例如，对 Windows 的文件备份程序，也许您只想列出最后一次备份后修改过的文件，像这样的文件具有档案字位集标志，因此您可以使用 DDL_EXCLUSIVE | DDL_ARCHIVE。

文件列表的排序

lParam 参数是指向文件说明串如"*.*"的指针，这个文件说明串不影响列表框中的子目录。

您也许希望给列有文件清单的列表框使用 LBS_SORT 消息。列表框首先列出符合文件说明的文件，再(可选)列出子目录名。列出的第一个子目录名将采用下面的格式：

[..]

这种"两个点"的子目录项允许用户向根目录返回一级。(在根目录下列出文件名时此项不会出现。)最后，具体的子目录名采用下面的形式：

[SUBDIR]

后面是以下面的形式列出的有效磁盘驱动器(也是可选的)：

[-A-]

9.6.7 Windows 的 head 程序

UNIX 中有一个著名的实用程序叫做 head，它显示文件的开始几行。让我们使用列表框为 Windows 编写一个类似的程序。如程序 9-6 所示，HEAD 将所有文件和子目录列在列表框中。您可以挑选某个选定的文件来显示，方法是使用鼠标双击该文件的文件名，或者当该文件名被选定时按下 Enter 键；您也可以使用这两种方法之一来改变子目录。这个程序在 HEAD 窗口客户区的右边，从文件的开头开始显示，它最多能够显示 8 KB 的内容。

程序 9-6 HEAD 程序

HEAD.C

```
/*-----------------------------------------
   HEAD.C -- Displays beginning (head) of file
             (c) Charles Petzold, 1998
  -----------------------------------------*/

#include <windows.h>
```

```
#define ID_LIST      1
#define ID_TEXT      2

#define MAXREAD      8192
#define DIRATTR      (DDL_READWRITE | DDL_READONLY | DDL_HIDDEN | DDL_SYSTEM | \
                      DDL_DIRECTORY | DDL_ARCHIVE | DDL_DRIVES)
#define DTFLAGS      (DT_WORDBREAK | DT_EXPANDTABS | DT_NOCLIP | DT_NOPREFIX)
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
LRESULT CALLBACK ListProc (HWND, UINT, WPARAM, LPARAM) ;

WNDPROC OldList ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("head") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) (COLOR_BTNFACE + 1) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("head"),
                          WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;
```

```
     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static BOOL        bValidFile ;
     static BYTE        buffer[MAXREAD] ;
     static HWND        hwndList, hwndText ;
     static RECT        rect ;
     static TCHAR       szFile[MAX_PATH + 1] ;
     HANDLE             hFile ;
     HDC                hdc ;
     int                i, cxChar, cyChar ;
     PAINTSTRUCT        ps ;
     TCHAR              szBuffer[MAX_PATH + 1] ;
     switch (message)
     {
     case WM_CREATE :
          cxChar = LOWORD (GetDialogBaseUnits ()) ;
          cyChar = HIWORD (GetDialogBaseUnits ()) ;

          rect.left = 20 * cxChar ;
          rect.top  =  3 * cyChar ;

          hwndList = CreateWindow (TEXT ("listbox"), NULL,
                              WS_CHILDWINDOW | WS_VISIBLE | LBS_STANDARD,
                              cxChar, cyChar * 3,
                              cxChar * 13 + GetSystemMetrics (SM_CXVSCROLL),
                              cyChar * 10,
                              hwnd, (HMENU) ID_LIST,
                              (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE),
                              NULL) ;

          GetCurrentDirectory (MAX_PATH + 1, szBuffer) ;

          hwndText = CreateWindow (TEXT ("static"), szBuffer,
                              WS_CHILDWINDOW | WS_VISIBLE | SS_LEFT,
                              cxChar, cyChar, cxChar * MAX_PATH, cyChar,
                              hwnd, (HMENU) ID_TEXT,
                              (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE),
                              NULL) ;
```

```
          OldList = (WNDPROC) SetWindowLong (hwndList, GWL_WNDPROC,
                                             (LPARAM) ListProc) ;

          SendMessage (hwndList, LB_DIR, DIRATTR, (LPARAM) TEXT ("*.*")) ;
          return 0 ;

     case WM_SIZE :
          rect.right  = LOWORD (lParam) ;
          rect.bottom = HIWORD (lParam) ;
          return 0 ;

     case WM_SETFOCUS :
          SetFocus (hwndList) ;
          return 0 ;

     case WM_COMMAND :
          if (LOWORD (wParam) == ID_LIST && HIWORD (wParam) == LBN_DBLCLK)
          {
               if (LB_ERR == (i = SendMessage (hwndList, LB_GETCURSEL, 0, 0)))
                    break ;

               SendMessage (hwndList, LB_GETTEXT, i, (LPARAM) szBuffer) ;

               if (INVALID_HANDLE_VALUE != (hFile = CreateFile (szBuffer,
                         GENERIC_READ, FILE_SHARE_READ, NULL,
                         OPEN_EXISTING, 0, NULL)))

               {
                    CloseHandle (hFile) ;
                    bValidFile = TRUE ;
                    lstrcpy (szFile, szBuffer) ;
                    GetCurrentDirectory (MAX_PATH + 1, szBuffer) ;

                    if (szBuffer [lstrlen (szBuffer) - 1] != '\\')
                         lstrcat (szBuffer, TEXT ("\\")) ;
                    SetWindowText (hwndText, lstrcat (szBuffer, szFile)) ;
               }
               else
               {
                    bValidFile = FALSE ;
                    szBuffer [lstrlen (szBuffer) - 1] = '\0' ;

                         // If setting the directory doesn't work, maybe it's
                         // a drive change, so try that.

                    if (!SetCurrentDirectory (szBuffer + 1))
                    {
```

```
                    szBuffer [3] = ':' ;
                    szBuffer [4] = '\0' ;
                    SetCurrentDirectory (szBuffer + 2) ;
                }

                    // Get the new directory name and fill the list box.

                GetCurrentDirectory (MAX_PATH + 1, szBuffer) ;
                SetWindowText (hwndText, szBuffer) ;
                SendMessage (hwndList, LB_RESETCONTENT, 0, 0) ;
                SendMessage (hwndList, LB_DIR, DIRATTR,
                                    (LPARAM) TEXT ("*.*")) ;
            }
            InvalidateRect (hwnd, NULL, TRUE) ;
        }
        return 0 ;
    case WM_PAINT :
        if (!bValidFile)
            break ;

        if (INVALID_HANDLE_VALUE == (hFile = CreateFile (szFile,
              GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL)))
        {
            bValidFile = FALSE ;
            break ;
        }

        ReadFile (hFile, buffer, MAXREAD, &i, NULL) ;
        CloseHandle (hFile) ;

            // i now equals the number of bytes in buffer.
            // Commence getting a device context for displaying text.

        hdc = BeginPaint (hwnd, &ps) ;
        SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
        SetTextColor (hdc, GetSysColor (COLOR_BTNTEXT)) ;
        SetBkColor  (hdc, GetSysColor (COLOR_BTNFACE)) ;

            // Assume the file is ASCII

        DrawTextA (hdc, buffer, i, &rect, DTFLAGS) ;

        EndPaint (hwnd, &ps) ;
        return 0 ;

    case WM_DESTROY :
        PostQuitMessage (0) ;
        return 0 ;
```

```
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}

LRESULT CALLBACK ListProc (HWND hwnd, UINT message,
                           WPARAM wParam, LPARAM lParam)
{
    if (message == WM_KEYDOWN && wParam == VK_RETURN)
        SendMessage (GetParent (hwnd), WM_COMMAND,
                     MAKELONG (1, LBN_DBLCLK), (LPARAM) hwnd) ;

    return CallWindowProc (OldList, hwnd, message, wParam, lParam) ;
}
```

在 ENVIRON 中，当我们选择一个环境变量时——无论是用鼠标还是用键盘——程序都将显示一个环境串。但是，如果我们在 HEAD 中使用这种选择显示方法，那么程序响应会很慢，这是因为在列表框中移动选择时，程序仍然要不断地打开和关闭文件。然而，HEAD 要求文件或者子目录被双击，从而引起一些问题，这是因为列表框控制没有鼠标双击的自动键盘接口。前面讲过，如果可能，应该尽量提供键盘接口。

解决的方法是什么呢？当然是窗口子类化。HEAD 中的列表框子类函数叫做 ListProc，它寻找 wParam 参数等于 VK_RETURN 的 WM_KEYDOWN 消息，并给其父窗口发送一个带 LBN_DBLCLK 通知码的 WM_COMMAND 消息。在 WndProc 中，对 WM_COMMAND 的处理使用了 Windows 函数的 CreateFile 来检查列表框中的选择。如果 CreateFile 返回一个错误信息，则表明该选择不是文件，而可能是一个子目录，然后 HEAD 使用 SetCurrentDirectory 来改变这个子目录。如果 SetCurrentDirectory 不能运行，程序将假定用户已经选择了一个驱动器字母。改变驱动器也需要调用 SetCurrentDirectory，除非需要避免使用短线而使用一个冒号。它向列表框发送一个 LB_RESETCONTENT 消息来清除其中的内容，再发送一个 LB_DIR 消息，使用新子目录中的文件来填充列表框。

WndProc 中的 WM_PAINT 消息是用 Windows 的 CreateFile 函数来打开文件的，这将返回一个文件句柄，该句柄可以传递给 Windows 的 ReadFile 和 CloseHandle 函数。

现在，我们解决本章开始部分涉及的问题：Unicode 问题。我们希望最好是让操作系统承认文本文件，这样 ReadFile 就能将 ASCII 文件转换成 Unicode 文本，或者将 Unicode 文件转换成 ASCII 文本。但事实并非如此。ReadFile 只读文件中未转换部分的字节，也就是说，DrawTextA（在可执行的编译中没有定义 UNICODE 标识符）会把文本解释为 ASCII，而 DrawTextW（Unicode 版）会假定文本是 Unicode。

因此程序应该能指出文件所包含的是 ASCII 文本还是 Unicode 文本，然后再恰当地调用 DrawTextA 或者 DrawTextW。实际上，HEAD 采用了更简单的方法，无论什么情况，它只调用 DrawTextA。

第十章

菜单及其他资源

大多数 Windows 程序都包含一个定制的图标，Windows 将该图标显示在应用程序窗口标题栏的左上角。当程序被列在“开始”菜单中，或者被显示在屏幕底部的任务栏中，或者被列在“Windows 资源管理器”中，或者作为快捷方式显示在桌面上时，Windows 也显示该程序的图标。有些程序——诸如 Windows Paint 一类的重要的图形绘制工具——也使用用户化的鼠标光标来表示程序的不同操作。许多 Windows 程序还使用菜单和对话框。菜单、对话框加上滚动条，这是标准 Windows 用户界面的面包和黄油。

图标、光标、菜单和对话框都是相互关联的，它们都是 Windows 的资源类型。资源即数据，它们被存储在程序的.EXE 文件中，但是它们并非驻留在程序的数据区域中。也就是说，资源不能通过程序源代码中定义的变量直接访问，Windows 提供函数直接或间接地把它们加载到内存中以备使用。我们已经遇到了两个这样的函数，即 LoadIcon 和 LoadCursor，它们出现在示例程序中定义窗口类结构的赋值语句中。它们从 Windows 中加载二进制图标和光标图像，并返回该图标或光标的句柄。在本章，我们从创建自己定制的图标开始，这些图标从该程序自己的.EXE 文件中加载。

在本书中，我们将讨论以下这些资源：

- 图标；
- 光标；
- 字符串；
- 定制资源；
- 菜单；
- 键盘加速键；
- 对话框；
- 位图。

前 6 个资源在本章讨论，对话框在第十一章讨论，位图在第十四章讨论。

10.1 图标、光标、字符串和定制资源

使用资源的好处之一在于程序的许多组件能够连编进程序的 .EXE 文件中。如果没有资源这一概念，例如图标图像的二进制文件可能会驻留在单独的文件中，.EXE 会把它读入内存中使用。或者图标不得不在程序中以字节数组的形式定义（这将不会看到实际的图标图像）。作为资源，图标存储在开发者计算机上的单独的可编辑文件中，但在编译过程中被连编进 .EXE 文件中。

10.1.1 将图标添加到程序

将资源添加到程序中需要 Visual C++ Developer Studio 的一些附加功能。对于图标来说，可以使用“Image Editor”（也称为“Graphics Editor”）来绘制图标的图像。该图像被保存在扩展名为 .ICO 的图标文件中。Developer Studio 还生成一个资源描述文件（扩展名为 .RC 的文件，有时也称作资源定义文件），它列出了程序的所有资源和一个让程序引用资源的头文件（RESOURCE.H）。

因此，您可以看到这些新文件是如何组织到一起的，让我们以创建名为 ICONDEMO 的新项目开始。像往常一样，在 Developer Studio 中从 File 菜单中选择 New，依次选择 Projects 选项卡和 Win32 Application。在 Project Name 域中键入 ICONDEMO，然后单击 OK。这时，Developer Studio 创建了用于支持工作区和项目的 5 个文件。这些文件包括文本文件 ICONDEMO.DSW、ICONDEMO.DSP 和 ICONDEMO.MAK（假定从 Tools 菜单选择 Open 后，在显示的 Open 对话框中，从 Build 选项卡中选中 Export makefile when saving project file。）

现在，让我们像通常那样创建 C 源代码文件。从 File 菜单中选择 New，选择 Files 选项卡，并单击 C++ Source File。在 File Name 域中键入 ICONDEMO.C，然后单击 OK。此时，Developer Studio 就创建了一个空的 ICONDEMO.C 文件。键入程序 10-1 中的程序，或选择 Insert 菜单，然后选中 File As Text 选项，从本书的配套光盘中复制源代码。

程序 10-1 ICONDEMO 程序

ICONDEMO.C

```
/*----------------------
   ICONDEMO.C -- Icon Demonstration Program
                 (c) Charles Petzold, 1998
  ----------------------*/

#include <windows.h>
```

```
#include "resource.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     TCHAR      szAppName[] = TEXT ("IconDemo") ;
     HWND       hwnd ;
     MSG        msg ;
     WNDCLASS   wndclass ;

     wndclass.style           = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc     = WndProc ;
     wndclass.cbClsExtra      = 0 ;
     wndclass.cbWndExtra      = 0 ;
     wndclass.hInstance       = hInstance ;
     wndclass.hIcon           = LoadIcon (hInstance, MAKEINTRESOURCE (IDI_ICON)) ;
     wndclass.hCursor         = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground   = GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName    = NULL ;
     wndclass.lpszClassName   = szAppName ;
     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Icon Demo"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
```

```
}
     static HICON  hIcon ;
     static int    cxIcon, cyIcon, cxClient, cyClient ;
     HDC           hdc ;
     HINSTANCE     hInstance ;
     PAINTSTRUCT   ps ;
     int           x, y ;

     switch (message)
     {
     case WM_CREATE :
          hInstance = ((LPCREATESTRUCT) lParam)->hInstance ;
          hIcon = LoadIcon (hInstance, MAKEINTRESOURCE (IDI_ICON)) ;
          cxIcon = GetSystemMetrics (SM_CXICON) ;
          cyIcon = GetSystemMetrics (SM_CYICON) ;
          return 0 ;

     case WM_SIZE :
          cxClient = LOWORD (lParam) ;
          cyClient = HIWORD (lParam) ;
          return 0 ;

     case WM_PAINT :
          hdc = BeginPaint (hwnd, &ps) ;

          for (y = 0 ; y < cyClient ; y += cyIcon)
               for (x = 0 ; x < cxClient ; x += cxIcon)
                    DrawIcon (hdc, x, y, hIcon) ;

          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

如果试图编译该程序，因为在程序开头引用的 RESOURCE.H 文件并不存在，所以会产生错误。然而，您不必直接创建 RESOURCE.H 文件，而是由 Developer Studio 为您创建一个。

您可以通过将资源描述文件添加到项目中来做到这一点。从 File 菜单中选择 New，选择 Files 选项卡，单击 Resource Script，在 File Name 域中键入 ICONDEMO，单击 OK。此时，Developer Studio 创建两个文本文件：ICONDEMO.RC（资源描述文件）和 RE-

SOURCE.H(允许 C 源代码文件和资源描述文件引用相同的已定义标识符)。不必直接编辑这两个文件,让 Developer Studio 来维护它们。如果您想查看资源描述文件和 RESOURCE.H,而不与 Developer Studio 产生干扰,可以用记事本程序打开它们。除非对您所做的很有把握,否则不要轻易地更改它们。请记住,只有当您有明确的操作命令或重新编译项目时,Developer Studio 才会保存这些文件的新版本。

资源描述文件是文本文件。它包括这些资源的可用文本形式表达的描述,例如菜单和对话框。资源描述文件也包括对非文本资源的二进制文件的引用,例如图标和自定义的鼠标光标。

现在,已经存在 RESOURCE.H 文件,您可以试着重新编译一下 ICONDEMO。现在会出现一条报错信息,指出 IDI_ICON 还没被定义。这个标识符第一次出现在下面的语句中:

```
wndclass.hIcon = LoadIcon (hInstance, MAKEINTRESOURCE (IDI_ICON)) ;
```

在本书前面的程序中这个语句是由下面的语句代替的:

```
wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION) ;
```

之所以改变语句是因为,以前我们为应用程序使用的是标准图标,而这里我们的目的是使用自定义图标。

那么让我们创建一个图标吧。在 Developer Studio 的 File View 窗口中,您会看到两个文件——ICONDEMO.C 和 ICONDEMO.RC。单击 CONDEMO.C 后,就可以编辑源代码。单击 ICONDEMO.RC 后,就可以把资源添加到文件中或编辑已存在的资源。要添加图标,从 Insert 菜单上选择 Resource。单击要添加的资源,也就是图标,然后单击 New 按钮。

现在呈现的是一个空白的 32 像素 × 32 像素的图标,可以向其中填充颜色。您会看到带有一组绘图工具和可用颜色的浮动的工具栏。注意颜色工具栏中包括两个与颜色无关的选项,它们有时指"屏幕"和"反屏幕"。当像素用"屏幕"着色时,它实际上是透明的。不管图标在什么表面上显示,图标未着色的部分会显示出底色。这样我们就可以创建非矩形的图标。

双击围绕图标的区域,会出现 Icon Properties 对话框,该对话框使您能够更改图标的 ID 和文件名。Developer Studio 可能已经将 ID 设置为 IDI_ICON1,将它改为 IDI_ICON,这样 ICONDEMO 就可以引用图标。(前缀 IDI 代表"图标的 ID"。)同样,将文件名改为 ICONDEMO.ICO。

现在选择一种颜色(如红色),并在图标上画一个大的 B(代表 BIG),不必像图 10-1 那么整齐。

此时程序应该能够编译并很好地运行。Developer Studio 将在 ICONDEMO.RC 资源

描述文件中划一条横线表示下面是带有标识符(IDI_ICON)的图标文件(ICONDEMO.ICO)。RESOURCE.H 头文件中包含 IDI_ICON 标识符的定义。

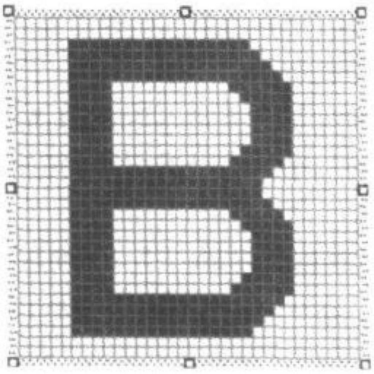


图 10-1 显示在 Developer Studio 中的标准(32×32) ICONDEMO 文件

Developer Studio 通过资源编译器 RC.EXE 编译资源。文本资源描述文件被转化为二进制形式,也就是具有扩展名.RES 的文件。然后,该已编译的资源文件随同.OBJ和.LIB 文件一起在 LINK 步骤中被指定。这就是资源被添加到最终的.EXE 文件中的方式。

当您运行 ICONDEMO 时,程序图标显示在标题栏的左上角和任务栏中。如果将程序添加到"开始"菜单中,或在桌面上添加了快捷方式,您也会在那儿看到该图标。

ICONDEMO 也在客户区水平和垂直地重复显示该图标。程序使用语句

```
hIcon = LoadIcon (hInstance, MAKEINTRESOURCE (IDI_ICON)) ;
```

获取图标的句柄。使用语句

```
cxIcon = GetSystemMetrics (SM_CXICON) ;
cyIcon = GetSystemMetrics (SM_CYICON) ;
```

获取图标的大小。然后,程序通过多次调用

```
DrawIcon (hdc, x, y, hIcon) ;
```

显示图标,其中 x 和 y 是被显示的图标的左上角的坐标。

在当前使用的大多数视频显示适配器上,带有 SM_CXICON 和 SM_CYICON 索引的 GetSystemMetrics 会报告图标的大小为 32 像素×32 像素。这是我们在 Developer Studio 中创建的图标大小,它也是图标出现在桌面上和显示在 ICONDEMO 程序客户区的大小。然而,这个大小并非显示在程序的标题栏或任务栏中的图标大小。小图标的大小可以由带有 SM_CXSMSIZE 和 SM_CYSMSIZE 索引的 GetSystemMetrics 获得。(第一个

SM 表示“system metrics(系统度量)”,被包含的 SM 表示“small(小)”。)对于当前使用的大多数显示适配器来说,小图标的大小为 16 像素 × 16 像素。

这就会产生问题。当 Windows 将 32 像素 × 32 像素的图标减小为 16 像素 × 16 像素的图标时,必须减少像素的行和列。这样,对于某些比较复杂的图标,就会产生失真。由于该原因,我们应该为那些图像缩小就会变形的图标创建特殊的 16 像素 × 16 像素的图标。在 Developer Studio 中图标图像的上面是标识为“Device”的组合框,在它的右边有一个按钮,单击该按钮会弹出 New Icon Image 对话框,选择 Small(16 像素 × 16 像素)。现在您可以画另一个图标。如图 10-2 所示,画一个“S”(表示“小”)。

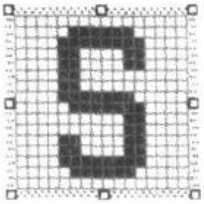


图 10-2 在 Developer Studio 中显示的小(16 × 16) ICONDEMO 文件

在该程序中您不必做任何事情。第二个图标图像被保存在相同的 ICONDEMO.ICO 文件中,并用相同的 INI_ICON 标识符引用。在适当的时候,Windows 会自动使用该较小的图标,例如在标题栏或任务栏中。当在桌面上显示快捷方式以及程序调用 DrawIcon 装饰客户区时,Windows 会使用大图标。

在掌握这些知识之后,让我们看一看使用图标的详细情况。

10.1.2 获取图标句柄

如果您仔细阅读 ICONDEMO.RC 和 RESOURCE.H 文件,会看到由 Developer Studio 生成的用于维护文件的一些标记。然而,当编译资源描述文件时,只有少数几行是重要的。这些从 ICONDEMO.RC 和 RESOURCE.H 文件中摘录下来的关键部分被列在程序 10-2 中。

程序 10-2 ICONDEMO.RC 和 RESOURCE.H 文件的摘录

ICONDEMO.RC(摘录)

```
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
// Icon
```

```
IDI_ICON                ICON        DISCARDABLE        "icondemo.ico"
```

RESOURCE.H(摘录)

```
// Microsoft Developer Studio generated include file.
// Used by IconDemo.rc

#define IDI_ICON                    101
```

程序 10-2 所示的 ICONDEMO.RC 和 RESOURCE.H 文件与您在普通的文本编辑器中手工创建的很相似，80 年代的 Windows 程序员就是这样做的。唯一不同的是 AFXRES.H，它是个头文件，包含了在创建由机器产生的 MFC 项目时由 Developer Studio 使用的常用标识符。在本书中，我们不会用到 AFXRES.H。

ICONDEMO.RC 中的这行

```
IDI_ICON ICON DISCARDABLE "icondemo.ico"
```

是资源描述文件的 ICON 语句。该图标有一个数字标识符 IDI_ICON，等于 101。由 Developer Studio 添加的 DISCARDABLE 关键字指出，必要时 Windows 可以从内存中丢弃图标，以获得额外的空间。之后不需要程序的任何特定操作，Windows 能够重新加载图标。DISCARDABLE 属性是默认的，不需要指定。只有当名称和目录路径包含空格时，Developer Studio 才将文件名放到引号中。

当资源编译程序将编译的资源保存在 ICONDEMO.RES 中，并且链接程序将资源添加到 ICONDEMO.EXE 中，该资源就仅通过一个资源类型(RT_ICON)和一个标识符(IDI_ICON 或 101)来标识。程序可以通过调用 LoadIcon 函数来获取此图标的句柄：

```
hIcon = LoadIcon (hInstance, MAKEINTRESOURCE (IDI_ICON)) ;
```

注意 ICONDEMO 在两个地方调用这个函数，一次在定义窗口类时，另一次在窗口过程中获取图标的句柄用于绘制。LoadIcon 返回 HICON 类型的值，它是图标的句柄。

LoadIcon 的第一个参数是指出资源来自哪个文件的实例句柄。使用 hInstance 表示它来自程序自己的.EXE 文件。LoadIcon 的第二个参数实际上被定义为指向字符串的指针。一会儿会看到，可以使用字符串而不是用数字标识符标识资源。宏 MAKEINTRESOURCE(把整数改制成资源串)生成指向非数字的指针，如下所示：

```
#define MAKEINTRESOURCE(i) (LPTSTR) ((DWORD) ((WORD) (i)))
```

LoadIcon 函数知道，如果第二个参数的高字为 0，那么低字就为图标的数字标识符。图标的标识符必须为 16 位值。

本书前面的示例程序使用了预先定义的图标：

```
LoadIcon (NULL, IDI_APPLICATION) ;
```

hInstance 参数被设定为 NULL,因此 Windows 知道这是预先定义的图标。IDI_APPLICATION 也在 WINUSER.H 中用 MAKEINTRESOURCE 定义:

```
#define IDI_APPLICATION MAKEINTRESOURCE(32512)
```

LoadIcon 的第二个参数带来了一个有趣的问题:图标的标识符可以为字符串吗?答案是可以,方法如下:在 Developer Studio 中,在 ICONDEMO 项目的文件列表中,选择 IDONDEMO.RC。您会看到顶端为 IconDemo Resource 的树型结构,然后是资源类型 Icon,再下来是 IDI_ICON。如果用鼠标右键单击图标标识符,并从菜单上选择 Properties,您就可以更改 ID。实际上,您可以把名称放在引号内将其更改为字符串。我用这种方法指定资源名称,并在本书的其他地方也使用该方法。

我喜欢为图标(以及一些其他资源)使用文本名称,因为名称可以是程序的名称。例如,假定文件被命名为 MYPROG。如果使用 Icon Properties 对话框将图标的 ID 指定为"MyProg"(包括引号),资源描述文件将包含下列语句:

```
MYPROG ICON DISCARDABLE myprog.ico
```

然而,在指出 MYPROG 是数字标识符的 RESOURCE.H 中将不包含 #define 语句。资源描述文件将假定 MYPROG 是字符串标识符。

在 C 程序中,使用 LoadIcon 函数来获取图标句柄。您可能已经有了表明程序名的字符串:

```
static TCHAR szAppName [] = TEXT ("MyProg") ;
```

这意味着程序可以使用语句

```
hIcon = LoadIcon (hInstance, szAppName) ;
```

来加载图标,这比宏 MAKEINTRESOURCE 更清晰一些。

但是如果您确想用数字来命名,那么可以用数字代替标识符或字符串。在 Icon Properties 对话框中,在 ID 域中输入数字。资源描述文件将有一个类似下面的 ICON 语句:

```
125 ICON DISCARDABLE myprog.ico
```

您可以使用两种方法之一引用图标。明显的方法是:

```
hIcon = LoadIcon (hInstance, MAKEINTRESOURCE (125)) ;
```

不明显的方法是:

```
hIcon = LoadIcon (hInstance, TEXT ("#125")) ;
```

Windows 识别初始字符 # 作为 ASCII 形式中数字的开头。

10.1.3 在程序中使用图标

虽然 Windows 以几种方式用图标来代表程序，但是许多 Windows 程序只有在用 WNDCLASS 结构和 RegisterClass 定义窗口类时才指定一个图标。如我们看到的，这样工作得很好，尤其当图标文件包含标准和较小的图像大小时，更是如此。Windows 在显示图标图像时，它会在图标文件中选择最合适的图像大小。

RegisterClass 有一个改进版本叫做 RegisterClassEx，它使用名为 WNDCLASSEX 的结构。WNDCLASSEX 有两个附加的字段：cbSize 和 hIconSm。cbSize 字段指出了 WNDCLASSEX 结构的大小，假设 hIconSm 被设置为小图标的图标句柄。这样，在 WNDCLASSEX 结构中，您可以设置与两个图标文件相关的两个图标句柄——一个用于标准图标，一个用于小图标。

有必要吗？没有。正如我们看到的，Windows 已经从单个的图标文件中提取了大小合适的图标图像。RegisterClassEx 似乎没有 RegisterClass 智能。如果 hIconSm 字段引用了包含多个图像的图标文件，则只有第一个图像能被利用。它可能是标准大小的图标，在大小上会被压缩。RegisterClassEx 似乎是为了使用多个图标图像而设计的，每个图像只包含一种图标大小。因为现在可以将多个图标大小包括在同一个图标文件中，所以我建议使用 WNDCLASS 和 RegisterClass。

如果您想在程序运行的时候，动态地更改程序的图标，可以使用 SetClassLong 来达到目的。例如，如果有与标识符 IDI_ALTICON 相关的第二个图标文件，则您可以使用以下语句将其切换到那个图标：

```
SetClassLong (hwnd, GCL_HICON,
    LoadIcon (hInstance, MAKEINTRESOURCE (IDI_ALTICON))) ;
```

如果不想保存程序图标的句柄，但要使用 DrawIcon 函数在别处显示它，可以使用 GetClassLong 获得句柄。例如：

```
DrawIcon (hdc, x, y, GetClassLong (hwnd, GCL_HICON)) ;
```

在 Windows 文档的某些部分，LoadIcon 被称为"过时的"，并推荐使用 LoadImage。(LoadIcon 在/Platform SDK/User Interface Services/Resources/Icons 中说明，LoadImage 在/Platform SDK/User Interface Services/Resources/Resources 中说明。)当然 LoadImage 更灵活，但它没有 LoadIcon 简单。您会注意到，在 ICONDEMO 中对同一个图标调用了 LoadIcon 两次。这不会产生问题，也没有使用额外的内存。LoadIcon 是获取句柄但不需要句柄被清除的少数几个函数之一。实际上有一个 DestroyIcon 函数，但它与 CreateIcon、CreateIconIndirect 和 CreateIconFromResource 连在一起使用。这些函数使程序能够动态地创建图标图像。

10.1.4 使用自定义光标

在程序中使用自定义的鼠标光标与使用自定义的图标相似，只是大多数程序员总是使用 Windows 提供的光标。自定义光标一般为单色，大小为 32 像素 × 32 像素。在 Developer Studio 中创建光标与创建图标的方法相同（从 Insert 菜单上选择 Resource，然后单击 Cursor），但不要忘记定义热点。

可以在类定义中设置自定义光标，语句为：

```
wndclass.hCursor = LoadCursor (hInstance, MAKEINTRESOURCE (IDC_CURSOR)) ;
```

如果光标用文本名称定义，则为：

```
wndclass.hCursor = LoadCursor (hInstance, szCursor) ;
```

每当鼠标位于根据这个类创建的窗口上时，就会显示与 IDC_CURSOR 或 szCursor 相关联的鼠标。

如果使用了子窗口，那么您可能希望光标随所在窗口的不同而有所区别。如果程序为这些子窗口定义了窗口类，就可以在每个窗口类中适当地设置 hCursor 字段，为每个类使用不同的光标。如果使用了预先定义的子窗口控制，就可以使用以下方法更改窗口类的 hCursor 字段：

```
SetClassLong (hwndChild, GCL_HCURSOR,
            LoadCursor (hInstance, TEXT ("childcursor")) ;
```

如果将客户区划分为较小的逻辑区域而不使用子窗口，就可以使用 SetCursor 来改变鼠标光标：

```
SetCursor (hCursor) ;
```

在处理 WM_MOUSEMOVE 消息期间，您应该调用 SetCursor；否则，当光标移动时，Windows 将使用窗口类中定义的光标来重画光标。文档指出，如果没有改变光标，则 SetCursor 速度很快。

10.1.5 字符串资源

最初涉及字符串资源的用户似乎觉得有些奇怪，使用源代码中定义为变量的常规字符串时，我们并没有任何问题。

字符串资源主要是为了使程序转换成其他语言的过程更为方便。正如后面两章中看到的那样，菜单和对话框也是资源描述文件的一部分。如果使用字符串资源而不是将字符串直接放入源代码中，那么程序使用的所有文本将在同一文件——资源描述文件中。如果转换了资源描述文件的文本，那么创建程序的另一种语言版本所需做的一

切就是重新链接程序。这种方法比重新组织源代码安全得多。(然而,除下一个示例程序外,我在本书的其他程序中不使用字符串表,原因是字符串表使代码看起来更模糊和复杂。)

通过在 Insert 菜单中选择 Resource,再选择 String Table,创建一个字符串表。字符串会显示在屏幕右边的列表中。通过双击字符串就可以选中它。对于每个字符串,您可以指定标识符和字符串的内容。

在资源描述中,字符串显示在一个多行的语句中,如下所示:

```
STRINGTABLE DISCARDABLE
BEGIN
     IDS_STRING1, "character string 1"
     IDS_STRING2, "character string 2"
     [other string definitions]
END
```

如果回到 Windows 编程的早期并在文本编辑器中手工创建这个字符串表(当然不是在 Developer Studio 中创建),您应该用左右大括弧代替 BEGIN 和 END 语句。

资源描述可以包含多个字符串表,但是每个 ID 必须唯一表示一个字符串。每个字符串占一行,最多 4097 个字符。能够将 \t 作为制表符, \n 用于行尾。这些控制符能够被 DrawText 和 MessageBox 函数识别。

您的程序可以使用 LoadString 调用把字符串赋值到程序数据段的缓冲区中去:

```
LoadString (hInstance, id, szBuffer, iMaxLength) ;
```

参数 id 是 ID 号,它加在资源描述文件中每个字符串的前面;szBuffer 是指向接收字符串的字符数组的指针;iMaxLength 是送入 szBuffer 中的最大字符数。函数返回字符串中的字符数。

每个字符串前面的 ID 号一般是定义在头文件中的宏标识符。许多 Windows 程序员使用前缀 IDS_来表示字符串的 ID 号。有时,文件名或其他信息需要在字符串显示时插入到字符串中,在这种情况下,您可以将 C 的格式化字符放入字符串,并把它用于 wsprintf 中作为一个格式化字符串。

所有资源文本——包括字符串表中的文本——以 Unicode 格式存储在.RES 编译资源文件及最终的.EXE 文件中。LoadStringW 函数直接加载 Unicode 文本。LoadStringA 函数(仅在 Windows 98 下有效)完成由 Unicode 到本地代码页的文本转换。

让我们来看一个程序,它使用三个字符串,在消息框中显示三条错误信息。RESOURCE.H 头文件为这些信息定义了三个标识符:

```
#define IDS_FILENOTFOUND  1
#define IDS_FILETOOBIG    2
#define IDS_FILEREADONLY  3
```

资源描述文件具有此字符串表：

```
STRINGTABLE
BEGIN
     IDS_FILENOTFOUND,       "File %s not found."
     IDS_FILETOOBIG,         "File %s too large to edit."
     IDS_FILEREADONLY,       "File %s is read-only."
END
```

C 源代码文件也包含这个头文件，并定义了一个显示消息框的函数。(我假定 szAppName 是一个包含程序名的全局变量。)

```
OkMessage (HWND hwnd, int iErrorNumber, TCHAR * szFileName)
{
     TCHAR szFormat [40] ;
     TCHAR szBuffer [60] ;

     LoadString (hInst, iErrorNumber, szFormat, 40) ;
     wsprintf (szBuffer, szFormat, szFilename) ;

     return MessageBox (hwnd, szBuffer, szAppName,
                        MB_OK | MB_ICONEXCLAMATION) ;
}
```

为了显示包含"file not found"信息的消息框，程序调用

```
OkMessage (hwnd, IDS_FILENOTFOUND, szFileName) ;
```

10.1.6 定制资源

Windows 还可以定义"定制资源"，也叫做"用户定义的资源"(用户就是您——程序员，而不是那个使用您的程序的幸运的人)。定制资源为连接.EXE 文件中的各种数据提供了方便，对获取程序中的数据也是如此。数据可以是您需要的任何格式。程序用于访问定制资源的 Windows 函数促使 Windows 将数据加载到内存中，并返回指向它的指针。然后您就可以对程序做任何操作。您会发现对于存储和访问各种私有数据，要比把数据存储在外部文件中再使用文件输入函数访问它要方便得多。

例如，您有一个文件叫做 BINDATA.BIN，它包含程序需要显示的一些数据。您可以选择这个文件的格式。如果在 MYPROG 项目中有 MYPROG.RC 资源描述文件，您就可以在 Developer Studio 中从 Insert 菜单中选择 Resource 并按下 Custom 按钮，来创建自定义的资源。键入表示资源的名称，例如 BINTYPE。然后，Developer Studio 会生成资源名(在这种情况下是 IDR_BINTYPE1)，并显示让您输入二进制数据的窗口。但是您不必输入什么，用鼠标右键单击 IDR_BINTYPE1 名，并选择 Properties，然后就可以输入一个文件名，例如 BINDATA.BIN。

资源描述文件就会包含以下的一行语句：

```
IDR_BINTYPE1 BINTYPE BINDATA.BIN
```

除了我们刚刚生成的 BINTYPET 资源类型外，这个语句与 ICONDEMO 中的 ICON 语句一样。有了图标后，您可以对资源名称使用文字的名称而不是数字的标识符。

当您编译并连接程序时，整个 BINDATA.BIN 文件会被连编到 MYPROG.EXE 文件中。

在程序的初始化（例如在处理 WM_CREATE 消息时）期间，您可以获得资源的句柄：

```
hResource = LoadResource (hInstance,
            FindResource (hInstance, TEXT ("BINTYPE"),
                          MAKEINTRESOURCE (IDR_BINTYPE1))) ;
```

变量 hResource 定义为 HGLOBAL 类型，它是指向内存块的句柄。不管它的名称是什么，LoadResource 不会立即将资源加载到内存中。联合使用 LoadResource 和 FindResource 函数实质上类似于 LoadIcon 和 LoadCursor 函数。事实上，LoadIcon 和 LoadCursor 函数用到了 LoadResource 和 FindResource 函数。

当您需要访问文本时，调用 LockResource：

```
pData = LockResource (hResource) ;
```

LockResource 将资源加载到内存中（如果还没有加载的话），然后它返回一个指向资源的指针。当结束对资源的使用时，您可以从内存中释放它：

```
FreeResource (hResource) ;
```

当您的程序终止时，也会释放资源，即使您没有调用 FreeResource。

让我们看一个使用三种资源——一个图标、一个字符串表和一个定制的资源——的示例程序。程序 10-3 所示的 POEPOEM 程序在其客户区显示 Edgar Allan Poe 的“Annabel Lee”文本。定制资源是文件 POEPOEM.TXT，它包含了诗文，此文本文件以反斜杠（ \ ）结束。

程序 10-3 POEPOEM 程序，包括一个图标和一个用户定义的资源

POEPOEM.C

```
/*-----------------------------------------
   POEPOEM.C -- Demonstrates Custom Resource
                (c) Charles Petzold, 1998
  -----------------------------------------*/

#include <windows.h>
```

```
#include "resource.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

HINSTANCE hInst ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     TCHAR      szAppName [16], szCaption [64], szErrMsg [64] ;
     HWND       hwnd ;
     MSG        msg ;
     WNDCLASS   wndclass ;

     LoadString (hInstance, IDS_APPNAME, szAppName,
                           sizeof (szAppName) / sizeof (TCHAR)) ;

     LoadString (hInstance, IDS_CAPTION, szCaption,
                           sizeof (szCaption) / sizeof (TCHAR)) ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (hInstance, szAppName) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          LoadStringA (hInstance, IDS_APPNAME, (char *) szAppName,
                                  sizeof (szAppName)) ;

          LoadStringA (hInstance, IDS_ERRMSG, (char *) szErrMsg,
                                  sizeof (szErrMsg)) ;
          MessageBoxA (NULL, (char *) szErrMsg,
                             (char *) szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, szCaption,
                          WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
```

```
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static char      * pText ;
     static HGLOBAL   hResource ;
     static HWND      hScroll ;
     static int       iPosition, cxChar, cyChar, cyClient, iNumLines, xScroll ;
     HDC              hdc ;
     PAINTSTRUCT      ps ;
     RECT             rect ;
     TEXTMETRIC       tm ;

     switch (message)
     {
     case WM_CREATE :
          hdc = GetDC (hwnd) ;
          GetTextMetrics (hdc, &tm) ;
          cxChar = tm.tmAveCharWidth ;
          cyChar = tm.tmHeight + tm.tmExternalLeading ;
          ReleaseDC (hwnd, hdc) ;

          xScroll = GetSystemMetrics (SM_CXVSCROLL) ;
          hScroll = CreateWindow (TEXT ("scrollbar"), NULL,
                              WS_CHILD | WS_VISIBLE | SBS_VERT,
                              0, 0, 0, 0,
                              hwnd, (HMENU) 1, hInst, NULL) ;

          hResource = LoadResource (hInst,
                      FindResource (hInst, TEXT ("AnnabelLee"),
                                           TEXT ("TEXT"))) ;

          pText = (char * ) LockResource (hResource) ;
          iNumLines = 0 ;

          while ( * pText != '\\' && * pText != '\0')
          {
```

```
               if ( * pText == '\n')
                    iNumLines ++ ;
               pText = AnsiNext (pText) ;
          }
          * pText = '\0' ;

          SetScrollRange (hScroll, SB_CTL, 0, iNumLines, FALSE) ;
          SetScrollPos   (hScroll, SB_CTL, 0, FALSE) ;
          return 0 ;

     case WM_SIZE :
          MoveWindow (hScroll, LOWORD (lParam) - xScroll, 0,
                      xScroll, cyClient = HIWORD (lParam), TRUE) ;
          SetFocus (hwnd) ;
          return 0 ;

     case WM_SETFOCUS :
          SetFocus (hScroll) ;
          return 0 ;

     case WM_VSCROLL :
          switch (wParam)
          {
          case SB_TOP :
               iPosition = 0 ;
               break ;
          case SB_BOTTOM :
               iPosition = iNumLines ;
               break ;
          case SB_LINEUP :
               iPosition -= 1 ;
               break ;
          case SB_LINEDOWN :
               iPosition += 1 ;
               break ;
          case SB_PAGEUP :
               iPosition -= cyClient / cyChar ;
               break ;
          case SB_PAGEDOWN :
               iPosition += cyClient / cyChar ;
               break ;
          case SB_THUMBPOSITION :
               iPosition = LOWORD (lParam) ;
               break ;
          }
          iPosition = max (0, min (iPosition, iNumLines)) ;
```

```
            if (iPosition != GetScrollPos (hScroll, SB_CTL))
            {
                SetScrollPos (hScroll, SB_CTL, iPosition, TRUE) ;
                InvalidateRect (hwnd, NULL, TRUE) ;
            }
            return 0 ;

        case WM_PAINT :
            hdc = BeginPaint (hwnd, &ps) ;

            pText = (char *) LockResource (hResource) ;

            GetClientRect (hwnd, &rect) ;
            rect.left += cxChar ;
            rect.top += cyChar * (1 - iPosition) ;
            DrawTextA (hdc, pText, -1, &rect, DT_EXTERNALLEADING) ;

            EndPaint (hwnd, &ps) ;
            return 0 ;

        case WM_DESTROY :
            FreeResource (hResource) ;
            PostQuitMessage (0) ;
            return 0 ;
        }
        return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

POEPOEM.RC(摘录)

```
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
// TEXT

ANNABELLEE              TEXT      DISCARDABLE       "poepoem.txt"

/////////////////////////////////////////////////////////////////////////////
// Icon

POEPOEM                 ICON      DISCARDABLE       "poepoem.ico"

/////////////////////////////////////////////////////////////////////////////
// String Table

STRINGTABLE DISCARDABLE
```

```
BEGIN
    IDS_APPNAME         "PoePoem"
    IDS_CAPTION         """Annabel Lee"" by Edgar Allan Poe"
    IDS_ERRMSG          "This program requires Windows NT!"
END
```

RESOURCE.H（摘录）

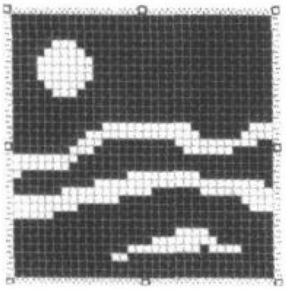
```
// Microsoft Developer Studio generated include file.
// Used by PoePoem.rc

#define IDS_APPNAME             1
#define IDS_CAPTION             2
#define IDS_ERRMSG              3
```

POEPOEM.TXT

```
It was many and many a year ago,
    In a kingdom by the sea,
That a maiden there lived whom you may know
    By the name of Annabel Lee;
And this maiden she lived with no other thought
    Than to love and be loved by me.

I was a child and she was a child
    In this kingdom by the sea,
But we loved with a love that was more than love ——
    I and my Annabel Lee ——
With a love that the winged seraphs of Heaven
    Coveted her and me.

And this was the reason that, long ago,
    In this kingdom by the sea,
A wind blew out of a cloud, chilling
    My beautiful Annabel Lee;
So that her highborn kinsmen came
    And bore her away from me,
To shut her up in a sepulchre
    In this kingdom by the sea.

The angels, not half so happy in Heaven,
    Went envying her and me ——
Yes! that was the reason (as all men know,
    In this kingdom by the sea)
That the wind came out of the cloud by night,
    Chilling and killing my Annabel Lee.

But our love it was stronger by far than the love
    Of those who were older than we ——
```

```
    Of many far wiser than we ——
And neither the angels in Heaven above
    Nor the demons down under the sea
Can ever dissever my soul from the soul
    Of the beautiful Annabel Lee:

For the moon never beams, without bringing me dreams
    Of the beautiful Annabel Lee;
And the stars never rise, but I feel the bright eyes
    Of the beautiful Annabel Lee:
And so, all the night-tide, I lie down by the side
Of my darling ——my darling ——my life and my bride,
    In her sepulchre there by the sea ——
    In her tomb by the sounding sea.

                    [May, 1849]
\
```

POEPOEM.ICO



在 POEPOEM.RC 资源描述文件中,用户定义的资源被定义为 TEXT 类型,取名为 AnnabelLee:

```
ANNABELLEE TEXT POEPOEM.TXT
```

在 WndProc 处理 WM_CREATE 时,使用 FindResource 和 LoadResource 获取资源句柄。使用 LockResource 锁定资源,并且使用一个小程序将文件末尾的反斜杠(\ )换成 0,这有利于后面 WM_PAINT 消息期间使用的 DrawText 函数。

注意,这里使用的是子窗口的滚动条,而不是窗口滚动条,这是因为子窗口滚动条有一个自动的键盘接口,因此在 POEPOEM 中没有使用 WM_KEYDOWN 处理。

POEPOEM 还使用三个字符串,它们的 ID 号在 RESOURCE.H 头文件中定义。在程序的开始,IDS_APPNAME 和 IDS_CAPTIONPOEPOEM 字符串由 LoadString 加载到内存中:

```
LoadString (hInstance, IDS_APPNAME, szAppName, sizeof (szAppName) /
```

```
                                          sizeof (TCHAR)) ;

LoadString (hInstance, IDS_CAPTION, szCaption, sizeof (szCaption) /
                                          sizeof (TCHAR)) ;
```

注意 RegisterClass 前部的两个调用。如果在 Windows 98 下运行 Unicode 版本的 POEPOEM,这两个调用就都会失败。因此,LoadStringA 比 LoadStringW 要复杂得多(LoadStringA 必须将资源字符串由 Unicode 转化为 ANSI,而 LoadStringW 直接加载它),LoadStringW 在 Windows 98 下不被支持。这意味着在 Windows 98 下,当 RegisterClassW 函数失败时,MessageBoxW 函数(Windows 98 支持)就不能使用 LoadStringW 加载程序的字符串。由于这个原因,程序使用 LoadStringA 加载 IDS_APPNAME 和 IDS_ERRMSG 字符串,并使用 MessageBoxA 显示自定义的消息框:

```
if (!RegisterClass (&wndclass))
{
     LoadStringA (hInstance, IDS_APPNAME, (char *) szAppName,
                         sizeof (szAppName)) ;

     LoadStringA (hInstance, IDS_ERRMSG, (char *) szErrMsg,
                         sizeof (szErrMsg)) ;

     MessageBoxA (NULL, (char *) szErrMsg,
                        (char *) szAppName, MB_ICONERROR) ;
     return 0 ;
}
```

注意,TCHAR 字符串变量是指向 char 的指针。

既然我们已经定义了用于 POEPOEM 的所有字符串资源,那么就很容易让翻译者将程序转换成外语版本。当然,它们将不得不翻译文本"Annabel Lee"——我想,这会是一项较为困难的工作。

10.2 菜单

您还记得 Monty Python 有关奶酪店的幽默短剧吗?是这么一回事:一个客人走进奶酪店想买某种奶酪。当然,店里没有这种奶酪。因此他又问有没有另一种奶酪,然后再问另一种,再问另一种,再问另一种(最后总共问了 40 种的奶酪),回答仍然是没有,没有,没有,没有,没有。

这个不幸的事件可以通过菜单的使用来避免。一个菜单是一列可用的选项,它告诉饥饿的就餐者厨房可以提供哪些服务,并且——对于 Windows 程序来说——还告诉用户应用程序能够执行哪些操作。

菜单可能是 Windows 程序提供的一致用户界面的最重要的部分,而在您的程序中

添加菜单是Windows编程中相对简单的部分。在Developer Studio中定义菜单。每个可选的菜单项被赋予唯一的ID号。在窗口类结构中指定菜单名。当用户选择一个菜单项时,Windows给您的程序发送包含该ID号的WM_COMMAND消息。

讨论完菜单后,我还将讨论键盘加速键,它们是一些键的组合,主要用于激活菜单功能。

10.2.1 菜单概念

窗口的菜单条紧接在标题栏的下方显示,这个菜单条有时被称为“主菜单”或“顶层菜单”。列在顶层菜单的项通常是下拉式菜单,也叫做“弹出式菜单”或“子菜单”。您也可以定义多重嵌套的弹出式菜单,也就是说,在弹出式菜单上的项可以访问另一个弹出式菜单。有时弹出式菜单上的项调用对话框以获得更多的信息(对话框在下一章介绍)。在标题栏的最左端,很多父窗口都显示程序的小图标,这个图标激活系统菜单,这实际上是另一个弹出式菜单。

弹出式菜单的各项可以是“被选中的”,这意味着Windows在菜单文本的左端显示一个小的复选标记,复选标记让用户知道从菜单中选中了哪些选项。这些选项之间可以是互斥的,也可以是不互斥的。顶层菜单项不能被选中。

顶层菜单或弹出式菜单项可以被“启用”、“禁用”或“灰化”。“激活”和“不激活”有时候被用作“启用”和“禁用”的同义词。标以启用或禁用的菜单项在用户看来是一样的,但是灰化的菜单项是使用灰色文本来显示的。

从用户的角度看,启用、禁用和灰化的菜单项都是可以“选择的”(加亮的),也就是说,用户可以使用鼠标选择被禁用的菜单项,或者将反显光标条移动到禁用的菜单项上,或者使用菜单项的关键字母来选择该菜单项。然而,从程序员的角度来看,启用、禁用和灰化菜单项的功能是不同的。Windows只为启用的菜单项向程序发送WM_COMMAND消息。对当前不合法的选项,可以禁用和灰化菜单项。如果想使用户知道选择是不合法的,那么您可以灰化它。

10.2.2 菜单结构

当您创建或更改程序中的菜单时,把顶层菜单和每一个弹出式菜单想象成各自独立的菜单是有用的。顶层菜单有一个菜单句柄,在顶层菜单中的每一个弹出式菜单也有它自己的菜单句柄。系统菜单(也是一个弹出式菜单)也有菜单句柄。

菜单中的每一项都有三个特性。第一个特性是菜单中显示什么,它可以是字符串或位图。第二个特性是WM_COMMAND消息中Windows发送给程序的菜单ID,或者是在用户选择菜单项时Windows显示的弹出式菜单的句柄。第三个特性是菜单项的属性,包括是否被禁用、灰化或被选中。

10.2.3 定义菜单

要使用 Developer Studio 来给程序资源描述文件添加菜单，可以从 Insert 菜单中选择 Resource 并选择 Menu。（或者您可能已经知道了。）然后，您可以交互式地定义菜单。菜单中每一项都有一个相关的 Menu Item Properties 对话框，指出项的文本串。如果选中 Pop-up 框，项就调用一个弹出式菜单，并且没有 ID 与之相联系。如果没有选中 Pop-up 框，该项就生成带有特定 ID 的 WM_COMMAND 消息。这两类菜单项分别出现在资源描述文件的 POPUP 和 MENUITEM 语句中。

当您为菜单中的项键入文字的时候，可以键入一个“&”符，指出后面一个字符在 Windows 显示菜单时要加下划线。这种下划线字符是在使用 Alt 键选择菜单项时 Windows 要查找的字符。如果在文本中不包括“&”符，就不显示下划线，Windows 会使用菜单项文本的第一个字母用于 Alt 键查找。

如果在 Menu Items Properties 对话框中复选 Grayed 选项，则菜单项是不能激活的，它的文本是灰色的，该项不产生 WM_COMMAND 消息。如果选中 Inactive 选项，则菜单项也是不能激活的，也不产生 WM_COMMAND 消息，但是它的文本显示正常。Checked 选项在菜单项边上放置一个复选标记。Separator 选项在弹出式菜单上产生一个分栏的横线。

在弹出式菜单的项上，可以在字符串中使用制表符 \t。紧接着 \t 的文本被放置在距弹出式菜单的第一列右边新的一列上。在本章后面，会看到在使用键盘加速键时它起的作用。字符串中的 \a 使跟着它的文本右对齐。

您指定的 ID 值是 Windows 发送给窗口过程中菜单消息中的数值。在菜单中 ID 值应该是唯一的。按照惯例，我使用以 IDM(“ID for a Menu”)开头的标识符。

10.2.4 在程序中引用菜单

大多数 Windows 应用程序在资源描述文件中只有一个菜单。您可以给菜单起一个与程序名相同文字的名称。程序员经常将程序名用于菜单名，以便相同的字符串可以用于窗口类、程序的图标名和菜单名。然后，程序在窗口的定义中为菜单引用该名称：

```
wndclass.lpszMenuName = szAppName ;
```

虽然访问菜单资源的最常用的方法是在窗口类中指定菜单，您也可以使用其他方法。Windows 应用程序可以使用 LoadMenu 函数将菜单资源加载到内存中，如同 LoadIcon 和 LoadCursor 函数一样。LoadMenu 返回一个菜单句柄。如果您在资源描述文件中为菜单使用了名称，语句如下：

```
hMenu = LoadMenu (hInstance, TEXT ("MyMenu")) ;
```

如果使用了数值，那么 LoadMenu 调用采用如下形式：

```
hMenu = LoadMenu (hInstance, MAKEINTRESOURCE (ID_MENU)) ;
```

然后，您可以将这个菜单句柄作为 CreateWindow 的第 9 个参数：

```
hwnd = CreateWindow (TEXT ("MyClass"), TEXT ("Window Caption"),
                     WS_OVERLAPPEDWINDOW,
                     CW_USEDEFAULT, CW_USEDEFAULT,
                     CW_USEDEFAULT, CW_USEDEFAULT,
                     NULL, hMenu, hInstance, NULL) ;
```

在这种情况下，CreateWindow 调用中指定的菜单可以覆盖窗口类中指定的任何菜单。如果 CreateWindow 的第 9 个参数是 NULL，那么您可以把窗口类中的菜单看作是基于这个窗口类的窗口默认菜单。这样，您可以为基于同一窗口类的几个窗口使用不同的菜单。

您也可以在窗口类中指定 NULL 菜单，并且在 CreateWindow 调用中也指定 NULL 菜单，然后在窗口被创建后再给窗口指定一个菜单：

```
SetMenu (hwnd, hMenu) ;
```

这种形式使您可以动态地修改窗口的菜单。在本章后面的 NOPOPUPS 程序中我们将会看到这方面的例子。

当窗口被清除时，同窗口相关的所有菜单都将被清除。同窗口不相关的菜单在程序结束前通过调用 DestroyMenu 被显式地清除。

10.2.5 菜单和消息

当用户选择一个菜单项时，Windows 通常向窗口过程发送几个不同的消息。在大多数情况下，您的程序可以忽略大部分消息，只需把它们传递给 DefWindowProc 即可。WM_INITMENU 就是这一类的消息，它具有下列参数：

wParam：	主菜单句柄
lParam：	0

wParam 值是主菜单句柄，即使用户选择的是系统菜单中的项。Windows 程序通常忽略 WM_INITMENU 消息。尽管在选中该项之前的消息已经给程序提供了修改菜单的机会，但是我们觉得此刻改变顶层菜单对用户来说是很为难的。

程序也接收 WM_MENUSELECT 消息。随着用户在菜单项中移动光标或者鼠标，程序会收到许多 WM_MENUSELECT 消息。这对实现那些包含对菜单项的文本描述的状态栏是很有帮助的。WM_MENUSELECT 的参数如下所示：

LOWORD (wParam): 选中项:菜单 ID 或者弹出式菜单句柄
HIWORD (wParam): 选择标志
lParam: 包含选中项的菜单句柄

WM_MENUSELECT 是一个菜单跟踪消息,wParam 的值告诉您当前选中的是菜单中的哪一项(加亮),wParam 的高位字中的"选择标志"可以是下列这些标志的组合:MF_GRAYED、MF_DISABLED、MF_CHECKED、MF_BITMAP、MF_POPUP、MF_HELP、MF_SYSMENU 和 MF_MOUSESELECT。如果需要根据对菜单项的选择来改变窗口客户区的内容,那么您可以使用 WM_MENUSELECT 消息。许多程序把该消息发送给 DefWindowProc。

当 Windows 准备显示一个弹出式菜单时,它给窗口过程发送一个 WM_INITMENUPOPUP 消息,参数如下:

wParam: 弹出式菜单句柄
LOWORD (lParam): 弹出式菜单索引
HIWORD (lParam): 系统菜单为 1,其他为 0

如果需要在显示弹出式菜单之前启用或者禁用菜单项,那么这个消息就很重要。例如,假定程序使用弹出式菜单上的 Paste 命令从剪贴板复制文本,那么当您收到弹出式菜单中的 WM_INITMENUPOPUP 消息时,应确定剪贴板内是否有文本存在。如果没有,则应该使 Paste 菜单项灰化。我们将在本章后面修改的 POPPAD 程序中看到这样的例子。

最重要的菜单消息是 WM_COMMAND,它表明用户已经从菜单中选中了一个被启用的菜单项。第八章中的 WM_COMMAND 消息也可以由子窗口控制产生。如果为菜单和子窗口控制使用同一 ID 码,那么您可以通过 lParam 的值来区别它们,菜单项的 lParam 值为 0,见表 10-1。

表 10-1

	菜单	控制
LOWORD (wParam):	菜单 ID	控制 ID
HIWORD (wParam):	0	通知码
lParam:	0	子窗口句柄

WM_SYSCOMMAND 消息类似于 WM_COMMAND 消息,只是 WM_SYSCOMMAND 表示用户从系统菜单中选择一个启用的菜单项:

wParam: 菜单 ID
lParam: 0

然而,如果 WM_SYSCOMMAND 消息是由按鼠标键产生的,LOWORD(lParam)和 HIWORD(lParam)将包含鼠标光标位置的 x 和 y 屏幕坐标。

对于 WM_SYSCOMMAND,菜单 ID 指示系统菜单中的哪一项被选中。对于预先定义的系统菜单项,应该通过和 0xFFF0 与操作屏蔽低 4 位,结果值应该为下列之一:SC_SIZE、SC_MOVE、SC_MINIMIZE、SC_MAXIMIZE、SC_NEXTWINDOW、SC_PREVWINDOW、SC_CLOSE、SC_VSCROLL、SC_HSCROLL、SC_ARRANGE、SC_RESTORE 和 SC_TASKLIST。此外,wParam 可以是 SC_MOUSEMENU 或 SC_KEYMENU。

如果在系统菜单中添加菜单项,那么 wParam 的低位字将是您定义的菜单 ID。为了避免与预先定义的菜单 ID 相冲突,应用程序应该使用小于 0xF000 的值,这对于将一般的 WM_SYSCOMMAND 消息发送给 DefWindowProc 是很重要的。如果不这样做,那么您实际上就是禁用了正常的系统菜单命令。

我们将讨论的最后一个消息是 WM_MENUCHAR,实际上它根本不是菜单消息。在下列两种情况之一发生时,Windows 会把这条消息发送到窗口过程:如果用户按 Alt 和一个与菜单项不匹配的字符时,或者当显示弹出式菜单而用户按下一个与弹出式菜单里的项不匹配的字符键时。随 WM_MENUCHAR 消息一起发送的参数如下所示:

LOWORD (wParam):	字符代码(ASCII 或 Unicode)
HIWORD (wParam):	选择码
lParam:	菜单句柄

选择码是:

- ◆ 0 不显示弹出式菜单;
- ◆ MF_POPUP 显示弹出式菜单;
- ◆ MF_SYSMENU 显示系统弹出式菜单。

Windows 程序通常把该消息传递给 DefWindowProc,它一般给 Windows 返回 0,这会使 Windows 发出蜂鸣。在第十四章 GRAFMENU 程序中将会看到 WM_MENUCHAR 消息的使用。

10.2.6 示例程序

让我们来看一个简单的例子。程序 10-4 所示的 MENUDEMO 程序在主菜单中有 5 项——File、Edit、Background、Timer 和 Help,每一项有一个弹出式菜单。MENUDEMO 只完成了最简单、最常用的菜单处理操作,包括捕获 WM_COMMAND 消息和检查 wParam 的低位字。

程序 10-4 MENUDEMO 程序

MENUDEMO.C

```
/*---------------------------------------
   MENUDEMO.C -- Menu Demonstration
                 (c) Charles Petzold, 1998
  ---------------------------------------*/

#include <windows.h>
#include "resource.h"

#define ID_TIMER 1

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

TCHAR szAppName[] = TEXT ("MenuDemo") ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     HWND     hwnd ;
     MSG      msg ;
     WNDCLASS wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = szAppName ;
     wndclass.lpszClassName = szAppName ;
     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Menu Demonstration"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
```

```
                            NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static int idColor [5] = { WHITE_BRUSH, LTGRAY_BRUSH, GRAY_BRUSH,
                                DKGRAY_BRUSH, BLACK_BRUSH } ;
     static int iSelection = IDM_BKGND_WHITE ;
     HMENU  hMenu ;

     switch (message)
     {
     case WM_COMMAND:
          hMenu = GetMenu (hwnd) ;

          switch (LOWORD (wParam))
          {
          case IDM_FILE_NEW:
          case IDM_FILE_OPEN:
          case IDM_FILE_SAVE:
          case IDM_FILE_SAVE_AS:
               MessageBeep (0) ;
               return 0 ;

          case IDM_APP_EXIT:
               SendMessage (hwnd, WM_CLOSE, 0, 0) ;
               return 0 ;

          case IDM_EDIT_UNDO:
          case IDM_EDIT_CUT:
          case IDM_EDIT_COPY:
          case IDM_EDIT_PASTE:
          case IDM_EDIT_CLEAR:
               MessageBeep (0) ;
               return 0 ;

          case IDM_BKGND_WHITE:              // Note: Logic below
```

```
case IDM_BKGND_LTGRAY:          //  assumes that IDM_WHITE
case IDM_BKGND_GRAY:            //  through IDM_BLACK are
case IDM_BKGND_DKGRAY:          //  consecutive numbers in
case IDM_BKGND_BLACK:           //  the order shown here.

    CheckMenuItem (hMenu, iSelection, MF_UNCHECKED) ;
    iSelection = LOWORD (wParam) ;
    CheckMenuItem (hMenu, iSelection, MF_CHECKED) ;

    SetClassLong (hwnd, GCL_HBRBACKGROUND, (LONG)
        GetStockObject
                (idColor [LOWORD (wParam) - IDM_BKGND_WHITE])) ;

    InvalidateRect (hwnd, NULL, TRUE) ;
    return 0 ;

case IDM_TIMER_START:
    if (SetTimer (hwnd, ID_TIMER, 1000, NULL))
    {
         EnableMenuItem (hMenu, IDM_TIMER_START, MF_GRAYED) ;
         EnableMenuItem (hMenu, IDM_TIMER_STOP,  MF_ENABLED) ;
    }
    return 0 ;

case IDM_TIMER_STOP:
    KillTimer (hwnd, ID_TIMER) ;
    EnableMenuItem (hMenu, IDM_TIMER_START, MF_ENABLED) ;
    EnableMenuItem (hMenu, IDM_TIMER_STOP,  MF_GRAYED) ;
    return 0 ;
case IDM_APP_HELP:
    MessageBox (hwnd, TEXT ("Help not yet implemented!"),
                szAppName, MB_ICONEXCLAMATION | MB_OK) ;
    return 0 ;

case IDM_APP_ABOUT:
    MessageBox (hwnd, TEXT ("Menu Demonstration Program\n")
                      TEXT ("(c) Charles Petzold, 1998"),
                szAppName, MB_ICONINFORMATION | MB_OK) ;
    return 0 ;
}
break ;

case WM_TIMER:
    MessageBeep (0) ;
    return 0 ;
```

```
        case WM_DESTROY:
            PostQuitMessage (0) ;
            return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

MENUDEMO.RC (摘录)

```
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
// Menu

MENUDEMO MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New",                IDM_FILE_NEW
        MENUITEM "&Open",               IDM_FILE_OPEN
        MENUITEM "&Save",               IDM_FILE_SAVE
        MENUITEM "Save &As...",         IDM_FILE_SAVE_AS
        MENUITEM SEPARATOR
        MENUITEM "E&xit",               IDM_APP_EXIT
    END
    POPUP "&Edit"
    BEGIN
        MENUITEM "&Undo",               IDM_EDIT_UNDO
        MENUITEM SEPARATOR
        MENUITEM "C&ut",                IDM_EDIT_CUT
        MENUITEM "&Copy",               IDM_EDIT_COPY
        MENUITEM "&Paste",              IDM_EDIT_PASTE
        MENUITEM "De&lete",             IDM_EDIT_CLEAR
    END
    POPUP "&Background"
    BEGIN
        MENUITEM "&White",              IDM_BKGND_WHITE, CHECKED
        MENUITEM "&Light Gray",         IDM_BKGND_LTGRAY
        MENUITEM "&Gray",               IDM_BKGND_GRAY
        MENUITEM "&Dark Gray",          IDM_BKGND_DKGRAY
        MENUITEM "&Black",              IDM_BKGND_BLACK
    END
    POPUP "&Timer"
    BEGIN
```

```
        MENUITEM "&Start",              IDM_TIMER_START
        MENUITEM "S&top",               IDM_TIMER_STOP, GRAYED
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&Help...",            IDM_APP_HELP
        MENUITEM "&About MenuDemo...",  IDM_APP_ABOUT
    END
END
```

RESOURCE.H (摘录)

```
// Microsoft Developer Studio generated include file.
// Used by MenuDemo.rc

#define IDM_FILE_NEW            40001
#define IDM_FILE_OPEN           40002
#define IDM_FILE_SAVE           40003
#define IDM_FILE_SAVE_AS        40004
#define IDM_APP_EXIT            40005
#define IDM_EDIT_UNDO           40006
#define IDM_EDIT_CUT            40007
#define IDM_EDIT_COPY           40008
#define IDM_EDIT_PASTE          40009
#define IDM_EDIT_CLEAR          40010
#define IDM_BKGND_WHITE         40011
#define IDM_BKGND_LTGRAY        40012
#define IDM_BKGND_GRAY          40013
#define IDM_BKGND_DKGRAY        40014
#define IDM_BKGND_BLACK         40015
#define IDM_TIMER_START         40016
#define IDM_TIMER_STOP          40017
#define IDM_APP_HELP            40018
#define IDM_APP_ABOUT           40019
```

MENUDEMO.RC 资源描述文件在定义菜单时会向您提供提示。菜单的文本名称为“MenuDemo”。大多数项有下划线字母，这就是说您必须在字母前键入“&”。MENUITEM SEPARATOR 语句是在 Menu Item Properties 对话框中选中 Separator 框产生的。注意菜单中有一个项具有 Checked 选项，另一个具有 Grayed 选项。还有，Background 弹出式菜单中的 5 个项应该按顺序输入，确保标识符按数值顺序排列，本程序需要这样。所有菜单项的标识符定义在 RESOURCE.H 中。

当收到弹出式菜单 File 和 Edit 各项有关的 WM_COMMAND 消息时，MENUDEMO 程序只使系统发出蜂鸣声。Background 弹出式菜单列出 MENUDEMO 用来给背景着色的 5

种现有画刷。在 MENUDEMO.RC 资源描述文件中，White 菜单项(菜单 ID 为 IDM_BKGND_WHITE)被标以 CHECKED，它在菜单项旁边设置复选标记。在MENUDEMO.C 中，iSelection 的值被初始化为 IDM_BKGND_WHITE。

Background 弹出式菜单上的 5 种画刷相互排斥。当 MENUDEMO.C 收到一个 WM_COMMAND 消息，而该消息中 wParam 是 Background 弹出式菜单中的 5 项之一时，它必须从先前选中的背景颜色中清除复选标记，并把标记添加到新的背景颜色上。为此，首先要得到菜单句柄：

```
hMenu = GetMenu (hwnd) ;
```

CheckMenuItem 函数用来取消当前选中的项：

```
CheckMenuItem (hMenu, iSelection, MF_UNCHECKED) ;
```

iSelection 的值被设置为 wParam 的值，新的背景颜色被选中：

```
iSelection = wParam ;
CheckMenuItem (hMenu, iSelection, MF_CHECKED) ;
```

窗口类中的背景颜色于是被替换为新的背景颜色，窗口客户区变为无效状态，Windows 使用新的背景颜色清除窗口。

Timer 弹出式菜单列出了两个选项——Start 和 Stop。开始时，Stop 选项被变灰(就像在资源描述文件中的菜单定义一样)。当您选择 Start 选项时，MENUDEMO 试图启动一个计时器，如果成功，则灰化 Start 选项，并激活 Stop 选项：

```
EnableMenuItem (hMenu, IDM_TIMER_START, MF_GRAYED) ;
EnableMenuItem (hMenu, IDM_TIMER_STOP, MF_ENABLED) ;
```

当收到一个 WM_COMMAND 消息，并且 wParam 等于 IDM_TIMER_STOP 时，MENUDEMO 程序停止计数，激活 Start 项，灰化 Stop 选项：

```
EnableMenuItem (hMenu, IDM_TIMER_START, MF_ENABLED) ;
EnableMenuItem (hMenu, IDM_TIMER_STOP, MF_GRAYED) ;
```

请注意，在计时器运行时，MENUDEMO 程序不可能收到 wParam 等于 IDM_TIMER_START 的 WM_COMMAND 消息。与此类似，在计时器关闭时，MENUDEMO 程序也不可能收到 wParam 等于 IDM_TIMER_STOP 的 WM_COMMAND 消息。

当 MENUDEMO 收到一个 WM_COMMAND 消息，而该消息的参数 wParam 等于 IDM_APP_ABOUT 或 IDM_APP_HELP 时，MENUDEMO 程序显示一个消息框(在下一章中，我们将把消息框变为对话框)。

当 MENUDEMO 程序收到一个 WM_COMMAND 消息，其参数 wParam 等于 IDM_APP

_EXIT 时，它给自己发送一个 WM_CLOSE 消息。这个消息与 DefWindowProc 收到 WM_SYSCOMMAND 消息，且 wParam 等于 SC_CLOSE 时发送给窗口过程的消息相同。我们将在本章后面介绍 POPPAD2 时再仔细研究这个问题。

10.2.7 菜单设计规范

在 MENUDEMO 中的 File 和 Edit 弹出式菜单的格式与其他 Windows 程序中的格式非常类似。Windows 的目的之一是为用户提供一种可识别的界面，而不要求用户为每个程序重新学习基本概念。如果 File 和 Edit 菜单在每个 Windows 程序中看起来都一样，并且都使用同样的字母和 Alt 键来进行选择，那么当然不会需要什么帮助了。

除了 File 和 Edit 弹出式菜单外，大多数 Windows 程序的菜单都是不同的。当设计一个菜单时，您应该看一看现有的 Windows 程序以尽量保持一致。当然，如果您认为别的程序是不对的，而您知道正确的方法，那么没有人能够阻止您。同时记住，修改一个菜单通常只需要修改资源描述文件，而不必修改程序代码。即使以后要改变菜单项的位置，也不会有多大的问题。

虽然您的程序菜单在顶层可以有 MENUITEM 语句，但这是不规范的，因为这样很容易导致错误的选择。如果您这样做，那么在文本串后面用一个惊叹号表示菜单项不会激活弹出式菜单。

10.2.8 定义菜单的复杂方法

在程序的资源描述文件中定义菜单通常是在窗口中添加菜单的最简单方法，但不是唯一的方法。如果您没有使用资源描述文件，那么可以使用 CreateMenu 和 AppendMenu 两个函数在程序中创建菜单。在定义完菜单后，您可以将菜单句柄发送给 CreateWindow，或者使用 SetMenu 来设置窗口的菜单。

以下是具体的做法。CreateMenu 简单地把一个句柄返回给新菜单：

```
hMenu = CreateMenu () ;
```

菜单一开始为空。AppendMenu 将菜单项插入菜单中。您必须为顶层菜单项和每一个弹出式菜单提供不同的菜单句柄。弹出式菜单是单独构成的，然后将弹出式菜单句柄插入顶层菜单。程序 10-5 中所示的代码就是用这种方法创建菜单的，实际上，这个菜单与 MENUDEMO 程序中的菜单相同。为了简化说明，代码使用 ASCII 字符串。

程序 10-5 不使用资源描述文件创建与 MENUDEMO 程序相同菜单的 C 代码

```
hMenu = CreateMenu () ;
```

```
hMenuPopup = CreateMenu () ;

AppendMenu (hMenuPopup, MF_STRING,      IDM_FILE_NEW,      "&New ") ;
AppendMenu (hMenuPopup, MF_STRING,      IDM_FILE_OPEN,     "&Open...") ;
AppendMenu (hMenuPopup, MF_STRING,      IDM_FILE_SAVE,     "&Save") ;
AppendMenu (hMenuPopup, MF_STRING,      IDM_FILE_SAVE_AS,  "Save &As...") ;
AppendMenu (hMenuPopup, MF_SEPARATOR, 0,                   NULL) ;
AppendMenu (hMenuPopup, MF_STRING,      IDM_APP_EXIT,      "E&xit") ;

AppendMenu (hMenu, MF_POPUP, hMenuPopup, "&File") ;

hMenuPopup = CreateMenu () ;

AppendMenu (hMenuPopup, MF_STRING,      IDM_EDIT_UNDO,     "&Undo") ;
AppendMenu (hMenuPopup, MF_SEPARATOR, 0,                   NULL) ;
AppendMenu (hMenuPopup, MF_STRING,      IDM_EDIT_CUT,      "Cu&t") ;
AppendMenu (hMenuPopup, MF_STRING,      IDM_EDIT_COPY,     "&Copy ") ;
AppendMenu (hMenuPopup, MF_STRING,      IDM_EDIT_PASTE,    "&Paste") ;
AppendMenu (hMenuPopup, MF_STRING,      IDM_EDIT_CLEAR,    "De&lete") ;
AppendMenu (hMenu, MF_POPUP, hMenuPopup, "&Edit") ;

hMenuPopup = CreateMenu () ;

AppendMenu (hMenuPopup, MF_STRING | MF_CHECKED, IDM_BKGND_WHITE,   "&White") ;
AppendMenu (hMenuPopup, MF_STRING,              IDM_BKGND_LTGRAY,  "&Light Gray ");
AppendMenu (hMenuPopup, MF_STRING,              IDM_BKGND_GRAY,    "&Gray ") ;
AppendMenu (hMenuPopup, MF_STRING,              IDM_BKGND_DKGRAY,  "&Dark Gray ");
AppendMenu (hMenuPopup, MF_STRING,              IDM_BKGND_BLACK,   "&Black") ;

AppendMenu (hMenu, MF_POPUP, hMenuPopup, "&Background") ;

hMenuPopup = CreateMenu () ;

AppendMenu (hMenuPopup, MF_STRING,              IDM_TIMER_START,   "&Start") ;
AppendMenu (hMenuPopup, MF_STRING | MF_GRAYED,  IDM_TIMER_STOP,    "S&top") ;

AppendMenu (hMenu, MF_POPUP, hMenuPopup, "&Timer ") ;

hMenuPopup = CreateMenu () ;

AppendMenu (hMenuPopup, MF_STRING, IDM_HELP_HELP,  "&Help") ;
AppendMenu (hMenuPopup, MF_STRING, IDM_APP_ABOUT,  "&About MenuDemo...") ;

AppendMenu (hMenu, MF_POPUP, hMenuPopup, "&Help") ;
```

我认为您会同意这个观点,即使用资源描述文件菜单模板更容易、更清楚一些。我并不鼓励您使用这里的方法定义菜单,而只是提供了一种实现菜单的方法。当然,您可以使用包含所有菜单项字符串、ID 和标志等的结构数组来压缩代码大小。不过,如果您这么做了,那么您还可以利用 Windows 定义菜单的第三种方法。LoadMenuIndirect 函数接受一个指向 MENUITEMTEMPLATE 类型的结构指针,并返回菜单的句柄,该函数在加载资源描述文件中的常规菜单模板后,在 Windows 中构造菜单。您不妨自己尝试一下。

10.2.9 浮动弹出式菜单

您还可以在没有顶层菜单条的情况下使用菜单,也就是说,您可以使弹出式菜单出现在屏幕顶层的任何位置。一种方法是使用鼠标右键来激活弹出式菜单。程序 10-6 所示的 POPMENU 说明了这种方法。

程序 10-6　POPMENU 程序

POPMENU.C

```
/*--------------------------------------------------
     POPMENU.C——Popup Menu Demonstration
                (c) Charles Petzold, 1998
   --------------------------------------------------*/

#include <windows.h>
#include "resource.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

HINSTANCE hInst ;
TCHAR     szAppName[] = TEXT ("PopMenu") ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     HWND       hwnd ;
     MSG        msg ;
     WNDCLASS   wndclass ;

     wndclass.style            = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc      = WndProc ;
     wndclass.cbClsExtra       = 0 ;
     wndclass.cbWndExtra       = 0 ;
     wndclass.hInstance        = hInstance ;
     wndclass.hIcon            = LoadIcon (NULL, szAppName) ;
```

```
     wndclass.hCursor         = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground   = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName    = NULL ;
     wndclass.lpszClassName   = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hInst = hInstance ;

     hwnd = CreateWindow (szAppName, TEXT ("Popup Menu Demonstration"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;
     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static HMENU hMenu ;
     static int   idColor [5] = { WHITE_BRUSH, LTGRAY_BRUSH, GRAY_BRUSH,
                                  DKGRAY_BRUSH, BLACK_BRUSH } ;
     static int   iSelection = IDM_BKGND_WHITE ;
     POINT        point ;

     switch (message)
     {
     case WM_CREATE:
          hMenu = LoadMenu (hInst, szAppName) ;
          hMenu = GetSubMenu (hMenu, 0) ;
          return 0 ;

     case WM_RBUTTONUP:
          point.x = LOWORD (lParam) ;
```

```
          point.y = HIWORD (lParam) ;
          ClientToScreen (hwnd, &point) ;

          TrackPopupMenu (hMenu, TPM_RIGHTBUTTON, point.x, point.y,
                          0, hwnd, NULL) ;
          return 0 ;

     case WM_COMMAND:
          switch (LOWORD (wParam))
          {
          case IDM_FILE_NEW:
          case IDM_FILE_OPEN:
          case IDM_FILE_SAVE:
          case IDM_FILE_SAVE_AS:
          case IDM_EDIT_UNDO:
          case IDM_EDIT_CUT:
          case IDM_EDIT_COPY:
          case IDM_EDIT_PASTE:
          case IDM_EDIT_CLEAR:
               MessageBeep (0) ;
               return 0 ;

          case IDM_BKGND_WHITE:         // Note: Logic below
          case IDM_BKGND_LTGRAY:        // assumes that IDM_WHITE
          case IDM_BKGND_GRAY:          // through IDM_BLACK are
          case IDM_BKGND_DKGRAY:        // consecutive numbers in
          case IDM_BKGND_BLACK:         // the order shown here.

               CheckMenuItem (hMenu, iSelection, MF_UNCHECKED) ;
               iSelection = LOWORD (wParam) ;
               CheckMenuItem (hMenu, iSelection, MF_CHECKED) ;

               SetClassLong (hwnd, GCL_HBRBACKGROUND, (LONG)
                    GetStockObject
                         (idColor [LOWORD (wParam) - IDM_BKGND_WHITE])) ;

               InvalidateRect (hwnd, NULL, TRUE) ;
               return 0 ;

          case IDM_APP_ABOUT:
               MessageBox (hwnd, TEXT ("Popup Menu Demonstration Program\n")
                                 TEXT ("(c) Charles Petzold, 1998"),
                           szAppName, MB_ICONINFORMATION | MB_OK) ;
               return 0 ;

          case IDM_APP_EXIT:
```

```
                SendMessage (hwnd, WM_CLOSE, 0, 0) ;
                return 0 ;

            case IDM_APP_HELP:
                MessageBox (hwnd, TEXT ("Help not yet implemented!"),
                            szAppName, MB_ICONEXCLAMATION | MB_OK) ;
                return 0 ;
            }
            break ;

        case WM_DESTROY:
            PostQuitMessage (0) ;
            return 0 ;
        }
        return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

POPMENU.RC (摘录)

```
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
// Menu

POPMENU MENU DISCARDABLE
BEGIN
    POPUP "MyMenu"
    BEGIN
        POPUP "&File"
        BEGIN
            MENUITEM "&New",                    IDM_FILE_NEW
            MENUITEM "&Open",                   IDM_FILE_OPEN
            MENUITEM "&Save",                   IDM_FILE_SAVE
            MENUITEM "Save &As",                IDM_FILE_SAVE_AS
            MENUITEM SEPARATOR
            MENUITEM "E&xit",                   IDM_APP_EXIT
        END
        POPUP "&Edit"
        BEGIN
            MENUITEM "&Undo",                   IDM_EDIT_UNDO
            MENUITEM SEPARATOR
            MENUITEM "Cu&t",                    IDM_EDIT_CUT
            MENUITEM "&Copy",                   IDM_EDIT_COPY
            MENUITEM "&Paste",                  IDM_EDIT_PASTE
            MENUITEM "De&lete",                 IDM_EDIT_CLEAR
```

```
        END
        POPUP "&Background"
        BEGIN
            MENUITEM "&White",                  IDM_BKGND_WHITE, CHECKED
            MENUITEM "&Light Gray",             IDM_BKGND_LTGRAY
            MENUITEM "&Gray",                   IDM_BKGND_GRAY
            MENUITEM "&Dark Gray",              IDM_BKGND_DKGRAY
            MENUITEM "&Black",                  IDM_BKGND_BLACK
        END
        POPUP "&Help"
        BEGIN
            MENUITEM "&Help...",                IDM_APP_HELP
            MENUITEM "&About PopMenu...",       IDM_APP_ABOUT
        END
    END
END
```

RESOURCE.H (摘录)

```
// Microsoft Developer Studio generated include file.
// Used by PopMenu.rc

#define IDM_FILE_NEW                    40001
#define IDM_FILE_OPEN                   40002
#define IDM_FILE_SAVE                   40003
#define IDM_FILE_SAVE_AS                40004
#define IDM_APP_EXIT                    40005
#define IDM_EDIT_UNDO                   40006
#define IDM_EDIT_CUT                    40007
#define IDM_EDIT_COPY                   40008
#define IDM_EDIT_PASTE                  40009
#define IDM_EDIT_CLEAR                  40010
#define IDM_BKGND_WHITE                 40011
#define IDM_BKGND_LTGRAY                40012
#define IDM_BKGND_GRAY                  40013
#define IDM_BKGND_DKGRAY                40014
#define IDM_BKGND_BLACK                 40015
#define IDM_APP_HELP                    40016
#define IDM_APP_ABOUT                   40017
```

资源描述文件 POPMENU.RC 定义的菜单同 MENUDEMO.RC 中的菜单非常相似,不同的是,在顶层菜单中只包含一项——一个弹出式菜单 MyMenu,它调用 File、Edit、Background 和 Help 选项。这 4 个选项垂直一列地出现在弹出式菜单上,而不是水平一行地出现在主菜单上。

在 WndProc 中的 WM_CREATE 处理期间,POPMENU 获取此弹出式菜单的句柄,就

是带有文本 MyMenu 的那个弹出式菜单：

```
hMenu = LoadMenu (hInst, szAppName) ;
hMenu = GetSubMenu (hMenu, 0) ;
```

在处理 WM_RBUTTONUP 消息期间，POPMENU 提供了鼠标指针的位置，将此位置转换为屏幕坐标，再将坐标值传递给 TrackPopupMenu：

```
point.x = LOWORD (lParam) ;
point.y = HIWORD (lParam) ;
ClientToScreen (hwnd, &point) ;

TrackPopupMenu (hMenu, TPM_RIGHTBUTTON, point.x, point.y,
                0, hwnd, NULL) ;
```

然后，Windows 显示出具有 File、Edit、Background 和 Help 项的弹出式菜单，选择其中任何一项都可以使嵌套的弹出式菜单显示在右边，菜单函数同通常的菜单一样。

如果要使用与该程序的主菜单相同的菜单并带有 TrackPopupMenu，您会遇到一些问题，因为函数需要弹出式菜单句柄。在"Microsoft Knowledge Base"文章 ID Q99806 中提供了一些信息。

10.2.10 使用系统菜单

使用 WS_SYSMENU 风格创建的父窗口在其标题栏的左侧有一个系统菜单框。如果您愿意，可以修改这个菜单。在 Windows 编程的早期，程序员一般把 About 菜单项放入系统菜单。虽然这种方法不常见，但是修改系统菜单往往是一种快而不当的在短程序中添加菜单的方法。这里唯一的限制是：在系统菜单中添加的命令的 ID 值必须小于 0xF000；否则它们将会同 Windows 系统菜单命令所使用的 ID 值相冲突。还要记住，当为这些新菜单项在窗口过程中处理 WM_SYSCOMMAND 消息时，您必须把其他的 WM_SYSCOMMAND 消息发送给 DefWindowProc。如果不这样做，那么实际上是禁用了系统菜单上的所有常规选项。

程序 10-7 中所示的 POORMENU（"不当的个人菜单"）在系统菜单中添加了一个分隔条和三个命令，最后一个命令将删除这些附加的菜单项。

程序 10-7 POORMENU 程序

POORMENU.C

```
/*----------------------------------------
   POORMENU.C——The Poor Person's Menu
              (c) Charles Petzold, 1998
  ----------------------------------------*/
```

```
#include <windows.h>

#define IDM_SYS_ABOUT    1
#define IDM_SYS_HELP     2
#define IDM_SYS_REMOVE   3

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

static TCHAR szAppName[] = TEXT ("PoorMenu") ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     HMENU      hMenu ;
     HWND       hwnd ;
     MSG        msg ;
     WNDCLASS   wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("The Poor-Person's Menu"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     hMenu = GetSystemMenu (hwnd, FALSE) ;

     AppendMenu (hMenu, MF_SEPARATOR, 0,             NULL) ;
     AppendMenu (hMenu, MF_STRING, IDM_SYS_ABOUT,    TEXT ("About...")) ;
     AppendMenu (hMenu, MF_STRING, IDM_SYS_HELP,     TEXT ("Help...")) ;
```

```
     AppendMenu (hMenu, MF_STRING, IDM_SYS_REMOVE,    TEXT ("Remove Additions")) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     switch (message)
     {
     case WM_SYSCOMMAND:
          switch (LOWORD (wParam))
          {
          case IDM_SYS_ABOUT:
               MessageBox (hwnd, TEXT ("A Poor-Person's Menu Program\n")
                                 TEXT ("(c) Charles Petzold, 1998"),
                           szAppName, MB_OK | MB_ICONINFORMATION) ;
               return 0 ;

          case IDM_SYS_HELP:
               MessageBox (hwnd, TEXT ("Help not yet implemented!"),
                           szAppName, MB_OK | MB_ICONEXCLAMATION) ;
               return 0 ;

          case IDM_SYS_REMOVE:
               GetSystemMenu (hwnd, TRUE) ;
               return 0 ;
          }
          break ;

     case WM_DESTROY:
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

三个菜单 ID 在 POORMENU.C 的开始部分定义：

```
#define IDM_ABOUT     1
#define IDM_HELP      2
```

```
# define IDM_REMOVE    3
```

在程序窗口创建之后,POORMENU 得到一个系统菜单的句柄:

```
hMenu = GetSystemMenu (hwnd, FALSE) ;
```

第一次调用 GetSystemMenu 时,您应该为修改菜单做准备,将第二个参数设置为 FALSE。

使用下列 4 个 AppendMenu 调用来实现对菜单的修改:

```
AppendMenu (hMenu, MF_SEPARATOR, 0,               NULL) ;
AppendMenu (hMenu, MF_STRING, IDM_SYS_ABOUT,      TEXT ("About...")) ;
AppendMenu (hMenu, MF_STRING, IDM_SYS_HELP,       TEXT ("Help...")) ;
AppendMenu (hMenu, MF_STRING, IDM_SYS_REMOVE,     TEXT ("Remove Additions"));
```

第一个 AppendMenu 调用是添加分隔条。选择 Remove Additions 菜单项将使得 POORMENU 删除这些附加的菜单项,这只要把第二个参数设置为 TRUE,再次调用 GetSystemMenu 即可:

```
GetSystemMenu (hwnd, TRUE) ;
```

标准系统菜单有下列选项:Restore、Move、Size、Minimize、Maximize 和 Close。它们产生 wParam 分别等于 SC_RESTORE、SC_MOVE、SC_SIZE、SC_MINIMUM、SC_MAXIMUM 和 SC_CLOSE 的 WM_SYSCOMMAND 消息。尽管 Windows 程序一般不这样做,但是您可以自己处理这些消息,而不把它们留给 DefWindowProc。您也可以使用下面所述的方法来禁用或删除系统菜单的标准选项。Windows 文档中还介绍了一些系统菜单的标准附加项,这些附加项使用标识符 SC_NEXTWINDOW、SC_PREVWINDOW、SC_VSCROLL、SC_HSCROLL 和 SC_ARRANGE。您也许会发现在一些应用程序中将这些命令添加到系统菜单中是合适的。

10.2.11 更改菜单

我们已经看到了如何使用 AppendMenu 函数为程序定义菜单,以及将菜单项添加到系统菜单中。在 Windows 3.0 之前,您不得不被迫使用 ChangeMenu 函数来完成这种工作。ChangeMenu 函数功能很多,在 Windows 的所有版本中它是最复杂的函数之一(至少在那个时候)。现在,许多函数比 ChangeMenu 函数还要复杂,并且 ChangeMenu 的功能被分解为 5 个新的函数:

- AppendMenu　在菜单尾部添加一个新的菜单项;
- DeleteMenu　删除菜单中一个现有的菜单项,并清除该项;
- InsertMenu　在菜单中插入一个新项;
- ModifyMenu　修改一个现有的菜单项;

◆ RemoveMenu　从菜单中删除某一项。

如果菜单项是一个弹出式菜单，那么 DeleteMenu 和 RemoveMenu 之间的区别就很重要。DeleteMenu 清除弹出式菜单，但 RemoveMenu 不清除它。

10.2.12 其他菜单命令

下面是在使用菜单时一些有用的函数。

当您改变顶层菜单项时，直到 Windows 重画菜单条时才显示所做的改变，您可以通过下列调用来强迫执行菜单刷新：

```
DrawMenuBar (hwnd) ;
```

注意，DrawMenuBar 的参数是窗口句柄而不是菜单句柄。

您可以使用下列命令来获得弹出式菜单的句柄：

```
hMenuPopup = GetSubMenu (hMenu, iPosition) ;
```

其中 iPosition 是 hMenu 指示的顶层菜单中弹出式菜单项的索引（开始为 0）。然后您可以在其他函数中使用弹出式菜单句柄（例如在 AppendMenu 函数中）。

您可以使用下列命令获得顶层菜单或者弹出式菜单中的当前项数：

```
iCount = GetMenuItemCount (hMenu) ;
```

您可以获取弹出式菜单项的菜单 ID：

```
id = GetMenuItemID (hMenuPopup, iPosition) ;
```

其中 iPosition 是菜单项在弹出式菜单中的位置（以 0 开始）。

在 MENUDEMO 中您已经看到如何选中或者取消选中弹出式菜单中的某一项：

```
CheckMenuItem (hMenu, id, iCheck) ;
```

在 MENUDEMO 中，hMenu 是顶层菜单的句柄，id 是菜单 ID，而 iCheck 的值是 MF_CHECKED 或 MF_UNCHECKED。如果 hMenu 是弹出式菜单句柄，那么参数 id 是位置索引，而不是菜单 ID。如果使用索引更方便的话，那么您可以在第三个参数中包含 MF_BYPOSITION，例如：

```
CheckMenuItem (hMenu, iPosition, MF_CHECKED | MF_BYPOSITION) ;
```

除了第三个参数是 MF_ENABLED、MF_DISABLED 或 MF_GRAYED 外，EnableMenuItem 函数与 CheckMenuItem 函数所完成的工作类似。如果您在具有弹出式菜单的顶层菜单项上使用 EnableMenuItem，那么必须在第三个参数中使用 MF_BYPOSITION 标识符，因为菜单项没有菜单 ID。我们将在本章后面所示的 POPPAD2 程序中看到 EnableMenu-

Item 的一个例子。HiliteMenuItem 也类似于 CheckMenuItem 和 EnableMenuItem,但是它使用的是 MF_HILITE 和 MF_UNHILITE。当您在菜单项之间移动时,Windows 使用反显方式加亮显示菜单项。通常不需要使用 HiliteMenuItem。

您还需要对菜单做些什么呢?您还记得我们在菜单中使用了一些什么字符串吗?您可以通过下面的调用来回顾一下:

```
iCharCount = GetMenuString (hMenu, id, pString, iMaxCount, iFlag) ;
```

iFlag 可以是 MF_BYCOMMAND(其中 id 是菜单 ID),也可以是 MF_BYPOSITION(其中 id 是位置索引)。函数将字符串的 iMaxCount 个字节复制到 pString 中,并返回复制的字节数。

或者您也许想知道菜单项的当前属性是什么:

```
iFlags = GetMenuState (hMenu, id, iFlag) ;
```

同样地,iFlag 可以是 MF_BYCOMMAND 或 MF_BYPOSITION。返回值 iFlags 是当前所有属性的组合,您可以通过对 MF_DISABLED、MF_GRAYED、MF_CHECKED、MF_MENUBREAK、MF_MENUBARBREAK 和 MF_SEPARATOR 标识符的检测来决定当前的属性。

也许现在您对菜单有了一些了解。这时您可能想知道如果您不再需要菜单时又应该如何处理。您可以使用下面的命令来清除菜单:

```
DestroyMenu (hMenu) ;
```

从而使菜单句柄无效。

10.2.13 创建菜单的非正规方法

现在让我们稍微偏离所讨论的主题。如果在您的程序中没有下拉式菜单,而是创建了多个没有弹出式菜单的顶层菜单,并调用 SetMenu 在顶层菜单之间切换,那会是什么样的情形呢?就像 Lotus 1-2-3 中老式的字符模式的菜单那样。程序 10-8 中的 NOPOPUPS 程序演示了如何处理这种情况。在这个程序中,File 和 Edit 项与 MENUDEMO 程序中的类似,但是却作为另一种顶层菜单显示出来。

程序 10-8 NOPOPUPS 程序

NOPOPUPS.C

```
/*-------------------------------------------------
   NOPOPUPS.C——Demonstrates No-Popup Nested Menu
               (c) Charles Petzold, 1998
  -------------------------------------------------*/
```

```
#include <windows.h>
#include "resource.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("NoPopUps") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName,
                          TEXT ("No-Popup Nested Menu Demonstration"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
```

```
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static HMENU hMenuMain, hMenuEdit, hMenuFile ;
     HINSTANCE    hInstance ;
     switch (message)
     {
     case WM_CREATE:
          hInstance = (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE) ;

          hMenuMain  = LoadMenu (hInstance, TEXT ("MenuMain")) ;
          hMenuFile  = LoadMenu (hInstance, TEXT ("MenuFile")) ;
          hMenuEdit  = LoadMenu (hInstance, TEXT ("MenuEdit")) ;

          SetMenu (hwnd, hMenuMain) ;
          return 0 ;

     case WM_COMMAND:
          switch (LOWORD (wParam))
          {
          case IDM_MAIN:
               SetMenu (hwnd, hMenuMain) ;
               return 0 ;

          case IDM_FILE:
               SetMenu (hwnd, hMenuFile) ;
               return 0 ;

          case IDM_EDIT:
               SetMenu (hwnd, hMenuEdit) ;
               return 0 ;

          case IDM_FILE_NEW:
          case IDM_FILE_OPEN:
          case IDM_FILE_SAVE:
          case IDM_FILE_SAVE_AS:
          case IDM_EDIT_UNDO:
          case IDM_EDIT_CUT:
          case IDM_EDIT_COPY:
          case IDM_EDIT_PASTE:
          case IDM_EDIT_CLEAR:
               MessageBeep (0) ;
               return 0 ;
          }
          break ;
```

```
          case WM_DESTROY:
               SetMenu (hwnd, hMenuMain) ;
               DestroyMenu (hMenuFile) ;
               DestroyMenu (hMenuEdit) ;

               PostQuitMessage (0) ;
               return 0 ;
          }
          return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

NOPOPUPS.RC (摘录)

```
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
// Menu

MENUMAIN MENU DISCARDABLE
BEGIN
    MENUITEM "MAIN:",                  0, INACTIVE
    MENUITEM "&File...",               IDM_FILE
    MENUITEM "&Edit...",               IDM_EDIT
END

MENUFILE MENU DISCARDABLE
BEGIN
    MENUITEM "FILE:",                  0, INACTIVE
    MENUITEM "&New",                   IDM_FILE_NEW
    MENUITEM "&Open...",               IDM_FILE_OPEN
    MENUITEM "&Save",                  IDM_FILE_SAVE
    MENUITEM "Save &As",               IDM_FILE_SAVE_AS
    MENUITEM "(&Main)",                IDM_MAIN
END

MENUEDIT MENU DISCARDABLE
BEGIN
    MENUITEM "EDIT:",                  0, INACTIVE
    MENUITEM "&Undo",                  IDM_EDIT_UNDO
    MENUITEM "Cu&t",                   IDM_EDIT_CUT
    MENUITEM "&Copy",                  IDM_EDIT_COPY
    MENUITEM "&Paste",                 IDM_EDIT_PASTE
    MENUITEM "De&lete",                IDM_EDIT_CLEAR
    MENUITEM "(&Main)",                IDM_MAIN
```

```
END
```

RESOURCE.H (摘录)

```
// Microsoft Developer Studio generated include file.
// Used by NoPopups.rc

#define IDM_FILE                    40001
#define IDM_EDIT                    40002
#define IDM_FILE_NEW                40003
#define IDM_FILE_OPEN               40004
#define IDM_FILE_SAVE               40005
#define IDM_FILE_SAVE_AS            40006
#define IDM_MAIN                    40007
#define IDM_EDIT_UNDO               40008
#define IDM_EDIT_CUT                40009
#define IDM_EDIT_COPY               40010
#define IDM_EDIT_PASTE              40011
#define IDM_EDIT_CLEAR              40012
```

在 Microsoft Developer Studio 中，您创建了三个菜单，而不是一个。从 Insert 中选择 Resource 三次，每个菜单有一个不同的名称。当窗口过程处理 WM_CREATE 消息时，Windows 将每个菜单资源加载到内存中：

```
hMenuMain = LoadMenu (hInstance, TEXT ("MenuMain")) ;
hMenuFile = LoadMenu (hInstance, TEXT ("MenuFile")) ;
hMenuEdit = LoadMenu (hInstance, TEXT ("MenuEdit")) ;
```

开始时，程序只显示主菜单：

```
SetMenu (hwnd, hMenuMain) ;
```

主菜单使用字符串 MAIN:、File...和 Edit...列出这三个选项。然而，MAIN:是禁用的，因此它不能使 WM_COMMAND 消息被发送到窗口过程。File 和 Edit 菜单项以 FILE:和 EDIT:开始，表明它们是子菜单。每个菜单的最后一项都是字符串(Main)，表示返回到主菜单。在这三个菜单之间进行切换很简单：

```
case WM_COMMAND :
     switch (wParam)
     {
     case IDM_MAIN :
          SetMenu (hwnd, hMenuMain) ;
          return 0 ;

     case IDM_FILE :
```

```
        SetMenu (hwnd, hMenuFile) ;
        return 0 ;

    case IDM_EDIT :
        SetMenu (hwnd, hMenuEdit) ;
        return 0 ;

    [other program lines]
    }
    break ;
```

在处理 WM_DESTROY 消息期间，NOPOPUPS 将程序的菜单设置为主菜单，并调用 DestroyMenu 来清除 File 和 Edit 菜单。当窗口被清除时，主菜单将自动被清除。

10.3 键盘加速键

加速键是产生 WM_COMMAND 消息(有些情况下是 WM_SYSCOMMAND)的键组合。许多时候，程序使用加速键来重复常用菜单项的动作(然而，加速键还可以用于执行非菜单功能)。例如，许多 Windows 程序在 Edit 菜单中都包含 Delete 或 Clear 选项，这些程序习惯上都将 Del 键指定为该选项的加速键。用户可以通过 Alt 键从菜单中选择 Delete 选项，或者只需按下加速键 Del。当窗口过程收到一个 WM_COMMAND 消息时，它不必确定使用的是菜单还是加速键。

10.3.1 为什么要使用加速键

您也许会问：为什么要使用加速键？为什么不能直接捕获 WM_KEYDOWN 或 WM_CHAR 消息而自己来实现同样的菜单功能呢？好处在哪里？对于一个单窗口应用程序，您当然可以捕获键盘消息，但是使用加速键可以得到一些好处：您不需要复制菜单和加速键逻辑。

对于有多个窗口和多个窗口过程的应用程序来说，加速键是非常重要的。正如我们所看到的，Windows 将键盘消息发送给当前活动窗口的窗口过程。然而对于加速键，Windows 把 WM_COMMAND 消息发送给窗口过程，该窗口过程的句柄在 Windows 函数 TranslateAccelerator 中给出。通常这是主窗口，也是拥有菜单的窗口，这意味着无需每个窗口过程都复制加速键的操作逻辑。

如果在主窗口的客户区中使用了非模态对话框(在下一章中讨论)或者子窗口，那么这种好处就变得非常重要。如果定义一个特定的加速键，以便在不同的窗口之间移动，那么只需要一个窗口过程有这个逻辑，子窗口不会收到加速键引发的 WM_COMMAND 消息。

10.3.2 安排加速键的几条规则

从理论上讲，您可以使用任何虚拟键或者字符键同 Shift 键、Ctrl 键或 Alt 键一起来定义加速键。然而，您应该尽力使得应用程序之间协调一致，并且尽量避免干扰 Windows 的键盘使用。在加速键中，应该避免使用 Tab、Enter、Esc 和 Spacebar 键，因为这些键常常用于完成系统功能。

加速键最常用的功能是操作程序 Edit 菜单中的各项。为这些菜单项推荐的加速键在 Windows 3.0 和 Windows 3.1 之间已有不同，因此通常都要支持如下表 10-2 所列的新旧两套加速键。

表 10-2

功能	旧加速键	新加速键
Undo	Alt + Backspace	Ctrl + Z
Cut	Shift + Del	Ctrl + X
Copy	Ctrl + Ins	Ctrl + C
Paste	Shift + Ins	Ctrl + V
Delete 或 Clear	Del	Del

另一种常用的虚拟键是激活帮助信息的功能键 F1。应该避免使用 F4、F5 和 F6 键，因为这些键常用在多文档界面(MDI)程序中完成特殊的功能(将在第十九章中讨论)。

10.3.3 加速键表

您可以在 Developer Studio 中定义加速键表。为了在程序中加载加速键表更容易，给它与程序名相同的名称(与菜单和图标名也相同)。

每个加速键都有在 Accel Properties 对话框中定义的 ID 和按键组合。如果您已经定义了菜单，则菜单 ID 会出现在组合框中，因此不用键入它们。

加速键可以是虚拟键代码，也可以是 ASCII 字符与 Shift、Ctrl 或 Alt 键的组合。您可以通过在字母前键入“^”来指定带有 Ctrl 键的 ASCII 字符，也可以从组合框中选取虚拟键。

为菜单项定义加速键时，应该将键的组合包含到菜单项的文本中。制表符(\ t)将文本与加速键分割开，将加速键列在第二列。为了在菜单中给加速键做标记，可以使用文本 Ctrl、Shift 或 Alt 之后跟上一个 + 号和一个键名(例如，Shift + F6 或 Ctrl + F6)。

10.3.4 加载加速键表

在您的程序中，使用 LoadAccelerators 函数来把加速键表加载到内存中，并获得该表的句柄。LoadAccelerators 语句非常类似于 LoadIcon、LoadCursor 和 LoadMenu 语句。

首先，把加速键表的句柄定义为类型 HANDLE：

```
HANDLE hAccel ;
```

然后加载加速键表：

```
hAccel = LoadAccelerators (hInstance, TEXT ("MyAccelerators")) ;
```

正如图标、光标和菜单一样，您可以使用一个数值代替加速键表的名称，然后在 LoadAccelerators 语句中和 MAKEINTRESOURCE 宏一起使用该数值，或者把它放在双引号内，前面冠以字符“#”。

10.3.5 键盘代码转换

现在我们将讨论这样的三行代码，在本书中到目前为止创建的所有 Windows 程序中都使用过它们。这些代码是标准的消息循环：

```
while (GetMessage (&msg, NULL, 0, 0))
{
     TranslateMessage (&msg) ;
     DispatchMessage (&msg) ;
}
```

下面修改选段代码，以便使用加速键：

```
while (GetMessage (&msg, NULL, 0, 0))
{
     if (!TranslateAccelerator (hwnd, hAccel, &msg))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
}
```

TranslateAccelerator 函数确定存放在 msg 消息结构中的消息是不是键盘消息。如果是，该函数将查找句柄为 hAccel 的加速键表。如果找到了一个匹配，则调用句柄为 hwnd 的窗口的窗口过程。如果加速键 ID 与系统菜单的菜单项一致，则消息就是 WM_SYSCOMMAND；否则，消息为 WM_COMMAND。

当 TranslateAccelerator 返回时，如果消息已经被转换（并且已经被发送给窗口过程），那么返回值为非 0；否则，返回值为 0。如果 TranslateAccelerator 返回一个非 0 值，则

不调用 TranslateMessage 和 DispatchMessage,而是经过循环回到 GetMessage 调用中。

TranslateMessage 中的参数 hwnd 看起来有点累赘,因为消息循环中的其他三个函数都没有要求这个参数。此外,消息结构本身(结构变量 msg)有一个叫做 hwnd 的成员,它是窗口句柄。

这是该函数不同的原因:msg 结构的域由 GetMessage 调用填充。当 GetMessage 的第二个参数为 NULL 时,函数检索应用程序所有窗口的消息。当 GetMessage 返回时,msg 结构的 hwnd 是将要获得消息的窗口的窗口句柄。然而,当 TranslateAccelerator 把键盘消息转换为 WM_COMMAND 或 WM_SYSCOMMAND 消息时,它使用函数的第一个参数指定的窗口句柄 hwnd 来代替窗口句柄 msg.hwnd。Windows 就是这样把所有加速键消息发送给同一窗口过程的,即使另一个应用窗口当前拥有输入焦点。当模态对话框或者消息框拥有输入焦点时,TranslateAccelerator 不转换键盘消息,因为这些窗口的消息是不经过程序的消息循环的。

在某些情况下,当您的程序的另一个窗口(比如一个非模态对话框)拥有输入焦点时,您也许不想转换加速键。您将在下一章中看到如何处理这种情况。

10.3.6 接收加速键消息

当加速键与系统菜单中的菜单项相对应时,TranslateAccelerator 给窗口过程发送一个 WM_SYSCOMMAND 消息,否则,TranslateAccelerator 给窗口过程发送一个 WM_COMMAND 消息。表 10-3 所示了几种可能接收到的 WM_COMMAND 消息,这些消息用于加速键、菜单命令,以及子窗口控制。

表 10-3

	加速键	菜单	控制
LOWORD (wParam)	加速键 ID	菜单 ID	控制 ID
HIWORD (wParam)	1	0	通知码
lParam	0	0	子窗口句柄

如果加速键与一个菜单项对应,那么窗口过程还会收到 WM_INITMENU、WM_INITMENUPOPUP 和 WM_MENUSELECT 消息,就好像选中了菜单选项一样。在处理 WM_INITMENUPOPUP 时,程序往往启用和禁用弹出式菜单中的菜单项,因此,在使用加速键时,您仍然能够实现这类功能。如果加速键与一个禁用或者灰化的菜单项相对应,那么,TranslateAccelerator 函数就不会向窗口过程发送 WM_COMMAND 或 WM_SYSCOMMAND 消息。

如果活动窗口已经被最小化,那么 TranslateAccelerator 将为与启用的系统菜单项相对应的加速键向窗口过程发送 WM_SYSCOMMAND 消息,而不是 WM_COMMAND 消息。

TranslateAccelerator 也会为没有任何菜单项与之对应的加速键发送该窗口过程的 WM_COMMAND 消息。

10.3.7 菜单与加速键应用程序 POPPAD

在第九章,我们创建了一个叫做 POPPAD1 的程序,它使用了子窗口编辑控制来实现基本的记事本功能。在这一章中,我们将添加 File 和 Edit 菜单,并称此程序为 POPPAD2。Edit 菜单的菜单项的功能全部可用;我们将在第十一章中完成 File 功能,在第十三章中完成 Print 功能。POPPAD2 如程序 10-9 所示。

程序 10-9 POPPAD2 程序

POPPAD2.C

```
/*------------------------
    POPPAD2.C——Popup Editor Version 2 (includes menu)
               (c) Charles Petzold, 1998
  -------------------------*/

#include <windows.h>
#include "resource.h"

#define ID_EDIT     1

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

TCHAR szAppName[] = TEXT ("PopPad2") ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     HACCEL     hAccel ;
     HWND       hwnd ;
     MSG        msg ;
     WNDCLASS   wndclass ;
     wndclass.style          = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc    = WndProc ;
     wndclass.cbClsExtra     = 0 ;
     wndclass.cbWndExtra     = 0 ;
     wndclass.hInstance      = hInstance ;
     wndclass.hIcon          = LoadIcon (hInstance, szAppName) ;
     wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground  = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName   = szAppName ;
     wndclass.lpszClassName  = szAppName ;
```

```
     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, szAppName,
                          WS_OVERLAPPEDWINDOW,
                          GetSystemMetrics (SM_CXSCREEN) / 4,
                          GetSystemMetrics (SM_CYSCREEN) / 4,
                          GetSystemMetrics (SM_CXSCREEN) / 2,
                          GetSystemMetrics (SM_CYSCREEN) / 2,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     hAccel = LoadAccelerators (hInstance, szAppName) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          if (!TranslateAccelerator (hwnd, hAccel, &msg))
          {
               TranslateMessage (&msg) ;
               DispatchMessage (&msg) ;
          }
     }
     return msg.wParam ;
}

AskConfirmation (HWND hwnd)
{
     return MessageBox (hwnd, TEXT ("Really want to close PopPad2?"),
                        szAppName, MB_YESNO | MB_ICONQUESTION) ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static HWND hwndEdit ;
     int         iSelect, iEnable ;

     switch (message)
     {
     case WM_CREATE:
          hwndEdit = CreateWindow (TEXT ("edit"), NULL,
```

```
                        WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL |
                        WS_BORDER | ES_LEFT | ES_MULTILINE |
                        ES_AUTOHSCROLL | ES_AUTOVSCROLL,
                        0, 0, 0, 0, hwnd, (HMENU) ID_EDIT,
                        ((LPCREATESTRUCT) lParam) -> hInstance, NULL) ;
          return 0 ;

     case WM_SETFOCUS:
          SetFocus (hwndEdit) ;
          return 0 ;

     case WM_SIZE:
          MoveWindow (hwndEdit, 0, 0, LOWORD (lParam), HIWORD (lParam), TRUE) ;
          return 0 ;

     case WM_INITMENUPOPUP:
          if (lParam == 1)
          {
               EnableMenuItem ((HMENU) wParam, IDM_EDIT_UNDO,
                    SendMessage (hwndEdit, EM_CANUNDO, 0, 0) ?
                                   MF_ENABLED : MF_GRAYED) ;

               EnableMenuItem ((HMENU) wParam, IDM_EDIT_PASTE,
                    IsClipboardFormatAvailable (CF_TEXT) ?
                                   MF_ENABLED : MF_GRAYED) ;

               iSelect = SendMessage (hwndEdit, EM_GETSEL, 0, 0) ;

               if (HIWORD (iSelect) == LOWORD (iSelect))
                    iEnable = MF_GRAYED ;
               else
                    iEnable = MF_ENABLED ;

               EnableMenuItem ((HMENU) wParam, IDM_EDIT_CUT,   iEnable) ;
               EnableMenuItem ((HMENU) wParam, IDM_EDIT_COPY,  iEnable) ;
               EnableMenuItem ((HMENU) wParam, IDM_EDIT_CLEAR, iEnable) ;
               return 0 ;
          }
          break ;
     case WM_COMMAND:
          if (lParam)
          {
               if (LOWORD (lParam) == ID_EDIT &&
                         (HIWORD (wParam) == EN_ERRSPACE ||
                          HIWORD (wParam) == EN_MAXTEXT))
                 MessageBox (hwnd, TEXT ("Edit control out of space."),
```

```
                         szAppName, MB_OK | MB_ICONSTOP) ;
          return 0 ;
     }
     else switch (LOWORD (wParam))
     {
     case IDM_FILE_NEW:
     case IDM_FILE_OPEN:
     case IDM_FILE_SAVE:
     case IDM_FILE_SAVE_AS:
     case IDM_FILE_PRINT:
          MessageBeep (0) ;
          return 0 ;

     case IDM_APP_EXIT:
          SendMessage (hwnd, WM_CLOSE, 0, 0) ;
          return 0 ;

     case IDM_EDIT_UNDO:
          SendMessage (hwndEdit, WM_UNDO, 0, 0) ;
          return 0 ;

     case IDM_EDIT_CUT:
          SendMessage (hwndEdit, WM_CUT, 0, 0) ;
          return 0 ;

     case IDM_EDIT_COPY:
          SendMessage (hwndEdit, WM_COPY, 0, 0) ;
          return 0 ;

     case IDM_EDIT_PASTE:
          SendMessage (hwndEdit, WM_PASTE, 0, 0) ;
          return 0 ;

     case IDM_EDIT_CLEAR:
          SendMessage (hwndEdit, WM_CLEAR, 0, 0) ;
          return 0 ;

     case IDM_EDIT_SELECT_ALL:
          SendMessage (hwndEdit, EM_SETSEL, 0, -1) ;
          return 0 ;

     case IDM_HELP_HELP:
          MessageBox (hwnd, TEXT ("Help not yet implemented!"),
                      szAppName, MB_OK | MB_ICONEXCLAMATION) ;
          return 0 ;
```

```
            case IDM_APP_ABOUT:
                MessageBox (hwnd, TEXT ("POPPAD2 (c) Charles Petzold, 1998"),
                            szAppName, MB_OK | MB_ICONINFORMATION) ;
                return 0 ;
            }
            break ;

        case WM_CLOSE:
            if (IDYES == AskConfirmation (hwnd))
                DestroyWindow (hwnd) ;
            return 0 ;

        case WM_QUERYENDSESSION:
            if (IDYES == AskConfirmation (hwnd))
                return 1 ;
            else
                return 0 ;

        case WM_DESTROY:
            PostQuitMessage (0) ;
            return 0 ;
        }
        return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

POPPAD2.RC (摘录)

```
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
// Icon

POPPAD2                 ICON        DISCARDABLE         "poppad2.ico"
/////////////////////////////////////////////////////////////////////////////
// Menu

POPPAD2 MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New",                    IDM_FILE_NEW
        MENUITEM "&Open...",                IDM_FILE_OPEN
        MENUITEM "&Save",                   IDM_FILE_SAVE
        MENUITEM "Save &As...",             IDM_FILE_SAVE_AS
        MENUITEM SEPARATOR
```

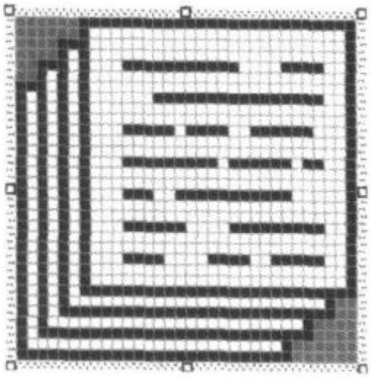
```
        MENUITEM "&Print",              IDM_FILE_PRINT
        MENUITEM SEPARATOR
        MENUITEM "E&xit",               IDM_APP_EXIT
    END
    POPUP "&Edit"
    BEGIN
        MENUITEM "&Undo\tCtrl+Z",       IDM_EDIT_UNDO
        MENUITEM SEPARATOR
        MENUITEM "Cu&t\tCtrl+X",        IDM_EDIT_CUT
        MENUITEM "&Copy\tCtrl+C",       IDM_EDIT_COPY
        MENUITEM "&Paste\tCtrl+V",      IDM_EDIT_PASTE
        MENUITEM "De&lete\tDel",        IDM_EDIT_CLEAR
        MENUITEM SEPARATOR
        MENUITEM "&Select All",         IDM_EDIT_SELECT_ALL
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&Help...",            IDM_HELP_HELP
        MENUITEM "&About PopPad2...",   IDM_APP_ABOUT
    END
END

/////////////////////////////////////////////////////////////////////////////
// Accelerator

POPPAD2 ACCELERATORS DISCARDABLE
BEGIN
    VK_BACK,        IDM_EDIT_UNDO,      VIRTKEY, ALT, NOINVERT
    VK_DELETE,      IDM_EDIT_CLEAR,     VIRTKEY, NOINVERT
    VK_DELETE,      IDM_EDIT_CUT,       VIRTKEY, SHIFT, NOINVERT
    VK_F1,          IDM_HELP_HELP,      VIRTKEY, NOINVERT
    VK_INSERT,      IDM_EDIT_COPY,      VIRTKEY, CONTROL, NOINVERT
    VK_INSERT,      IDM_EDIT_PASTE,     VIRTKEY, SHIFT, NOINVERT
    "^C",           IDM_EDIT_COPY,      ASCII, NOINVERT
    "^V",           IDM_EDIT_PASTE,     ASCII, NOINVERT
    "^X",           IDM_EDIT_CUT,       ASCII, NOINVERT
    "^Z",           IDM_EDIT_UNDO,      ASCII, NOINVERT
END
```

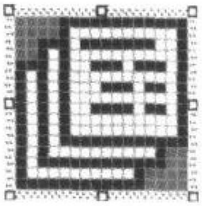
RESOURCE.H (摘录)

```
// Microsoft Developer Studio generated include file.
// Used by POPPAD2.RC

#define IDM_FILE_NEW                    40001
#define IDM_FILE_OPEN                   40002
```

```
#define IDM_FILE_SAVE            40003
#define IDM_FILE_SAVE_AS         40004
#define IDM_FILE_PRINT           40005
#define IDM_APP_EXIT             40006
#define IDM_EDIT_UNDO            40007
#define IDM_EDIT_CUT             40008
#define IDM_EDIT_COPY            40009
#define IDM_EDIT_PASTE           40010
#define IDM_EDIT_CLEAR           40011
#define IDM_EDIT_SELECT_ALL      40012
#define IDM_HELP_HELP            40013
#define IDM_APP_ABOUT            40014
```

POPPAD2.ICO





POPPAD2.RC 资源描述文件包含菜单和加速键。您将注意到，所有加速键都表示在制表符(\t)后的 Edit 弹出式菜单的字符串中。

10.3.8 启用菜单项

窗口过程的工作包括启用和灰化 Edit 菜单中的选项，这项工作在处理 WM_INITMENUPOPUP 时完成。首先，程序检查是否要显示 Edit 弹出式菜单。因为菜单里 Edit 的位置索引(File 从 0 开始)是 1，因此如果即将显示 Edit 弹出式菜单，那么 lParam 应该等于 1。

为了确定是否启用 Undo 选项，POPPAD2 给编辑控制发送一个 EM_CANUNDO 消息。如果编辑控制能够执行 Undo 动作，那么 SendMessage 调用返回非 0 值。在这种情况下，选项被启用；否则，选项灰化：

```
EnableMenuItem (wParam, IDM_UNDO,
    SendMessage (hwndEdit, EM_CANUNDO, 0, 0) ?
```

```
                    MF_ENABLED : MF_GRAYED) ;
```

只有当剪贴板中包含文本时，Paste 选项才能够被启用。我们可以使用 CF_TEXT 标识符通过 IsClipboardFormatAvailable 调用来确定这一点：

```
EnableMenuItem (wParam, IDM_PASTE,
    IsClipboardFormatAvailable (CF_TEXT) ? MF_ENABLED : MF_GRAYED) ;
```

只有选择了编辑控制中的文本，Cut、Copy 和 Delete 选项才能够被启用。给编辑控制发送一个 EM_GETSEL 消息，并返回包含此信息的整数：

```
iSelect = SendMessage (hwndEdit, EM_GETSEL, 0, 0) ;
```

iSelect 的低位字是第一个选中字符的位置，iSelect 的高位字是下一个选中字符的位置。如果这两个字相等，则表明没有选中文本：

```
if (HIWORD (iSelect) == LOWORD (iSelect))
      iEnable = MF_GRAYED ;
else
      iEnable = MF_ENABLED ;
```

然后可以将 iEnable 的值用于 Cut、Copy 和 Delete 选项：

```
EnableMenuItem (wParam, IDM_CUT,  iEnable) ;
EnableMenuItem (wParam, IDM_COPY, iEnable) ;
EnableMenuItem (wParam, IDM_DEL,  iEnable) ;
```

10.3.9 处理菜单项

当然，如果对 POPPAD2 程序不使用子窗口编辑控制，那么我们将面临一些问题，这涉及如何完成 Edit 菜单中的 Undo、Cut、Copy、Paste、Clear 和 Select All 选项。正是编辑控制使得这种处理变得容易，因为我们对每一个选项只需向编辑控制发送一个消息即可：

```
case IDM_UNDO :
    SendMessage (hwndEdit, WM_UNDO, 0, 0) ;
    return 0 ;

case IDM_CUT :
    SendMessage (hwndEdit, WM_CUT, 0, 0) ;
    return 0 ;

case IDM_COPY :
    SendMessage (hwndEdit, WM_COPY, 0, 0) ;
    return 0 ;
```

```
case IDM_PASTE :
    SendMessage (hwndEdit, WM_PASTE, 0, 0) ;
    return 0 ;

case IDM_DEL :
    SendMessage (hwndEdit, WM_DEL, 0, 0) ;
    return 0 ;

case IDM_SELALL :
    SendMessage (hwndEdit, EM_SETSEL, 0, -1) ;
    return 0 ;
```

注意，我们可以进一步简化这些处理——只要使 IDM_UNDO、IDM_CUT 等的值等于相应的窗口消息 WM_UNDO、WM_CUT 等的值。

File 弹出式菜单上的 About 选项激活一个简单的消息框：

```
case IDM_ABOUT :
    MessageBox (hwnd, TEXT ("POPPAD2 (c) Charles Petzold, 1998"),
                szAppName, MB_OK | MB_ICONINFORMATION) ;
    return 0 ;
```

在下一章中，我们将把它变成一个对话框。当您从菜单中选择 Help 选项或者按下 F1 加速键时，同样可以激活一个消息框。

Exit 选项向窗口过程发送一个 WM_CLOSE 消息：

```
case IDM_EXIT :
    SendMessage (hwnd, WM_CLOSE, 0, 0) ;
    return 0 ;
```

这正是 DefWindowProc 收到一个 wParam 等于 SC_CLOSE 的 WM_SYSCOMMAND 消息时所完成的工作。

在前面的那些程序中，我们没有在窗口过程中处理 WM_CLOSE 消息，而只是简单地把它发送给 DefWindowProc。DefWindowProc 对 WM_CLOSE 的处理非常简单：调用 DestroyWindow 函数。可以不把 WM_CLOSE 消息发送给 DefWindowProc，而让 POPPAD2 来处理它。这个事实现在并不重要，但是在第十一章中当 POPPAD 可以真正编辑文本时，它就变得非常重要了。

```
case WM_CLOSE :
    if (IDYES == AskConfirmation (hwnd))
        DestroyWindow (hwnd) ;
    return 0 ;
```

AskConfirmation 是 POPPAD2 中的一个函数，它显示一个请求确认关闭程序的消息框：

```
AskConfirmation (HWND hwnd)
{
     return MessageBox (hwnd, TEXT ("Really want to close Poppad2?"),
                        szAppName, MB_YESNO | MB_ICONQUESTION) ;
}
```

消息框(以及 AskConfirmation 函数)返回 IDYES,如果选择了 Yes 按钮的话。只有这样,程序才调用 DestroyWindow,否则,程序并不结束。

如果在程序结束之前想确认某种操作,那么您还必须处理 WM_QUERYENDSESSION 消息。当用户结束一个 Windows 执行时,Windows 开始向每个窗口过程发送一个 WM_QUERYENDSESSION 消息。如果有任何一个窗口过程从这个消息中返回 0,那么 Windows 将不会结束。我们如下处理 WM_QUERYENDSESSION:

```
case WM_QUERYENDSESSION :
    if (IDYES == AskConfirmation (hwnd))
        return 1 ;
    else
        return 0 ;
```

如果在程序结束之前要求用户的确认,必须处理 WM_CLOSE 和 WM_QUERYENDSESSION 这两个消息,这就是为什么我们使 POPPAD2 中的 Exit 菜单选项只向窗口过程发送一个 WM_CLOSE 消息的原因。这样做,我们避免了在第三处进行请求确认的操作。

如果要处理 WM_QUERYENDSESSION 消息,那么您也许还会对 WM_ENDSESSION 消息感兴趣。Windows 把这个消息发送给先前收到 WM_QUERYENDSESSION 消息的每个窗口过程。如果由于另一个程序从 WM_QUERYENDSESSION 返回了 0 而不能结束 Windows 的运行,那么 WM_ENDSESSION 的 wParam 参数为 0。WM_ENDSESSION 消息实际上回答了这个问题:我告诉过 Windows 可以结束,但是真的结束了吗?

尽管在 POPPAD2 的 File 菜单中我包括了常见的 New、Open、Save 和 Save As 选项,但是它们现在并不起作用。要处理这些命令,我们需要使用对话框。现在是讨论对话框的时候了。也是您准备学习它们的时候了。

第十一章

对话框

如果有很多输入内容，从而超出了菜单可以处理的程度，那么我们可以使用对话框来获取输入。程序员可以通过在某项后面加上省略号(…)来表示该菜单项将激活一个对话框。

对话框的一般形式是包含多种子窗口控制的弹出式窗口，这些控制的大小和位置在程序资源描述文件的“对话框模板”中指定。虽然程序员能够“手工”定义对话框模板，但是现在通常是在 Visual C++ Developer Studio 中交互设计的。然后由 Developer Studio 创建对话框模板。

当程序调用基于模板的对话框时，Microsoft Windows 98 负责创建弹出式对话框窗口和子窗口控制，并提供处理对话框消息(包括所有键盘和鼠标输入)的窗口过程。有时候称完成这些功能的 Windows 内部代码为“对话框管理器”。

Windows 的内部对话框窗口过程所处理的许多消息也传递给您自己程序中的函数，这个函数即是所谓的“对话框过程”或者“对话过程”。对话过程与通常的窗口过程类似，但是也存在一些重要区别。一般说来，除了在创建对话框时初始化子窗口控制，处理来自子窗口控制的消息以及结束对话框之外，程序员不需要再给对话框过程增加其他功能。对话过程通常不处理 WM_PAINT 消息，也不直接处理键盘和鼠标输入。

对话框这个主题的含义太广了，因为它还包含子窗口控制的使用。不过，我们已经在第九章研究了子窗口控制。当您在对话框中使用子窗口控制时，第九章所提到的许多工作都可以由 Windows 的对话框管理器来完成。尤其是在程序 COLORS1 中遇到的滚动条之间输入焦点的传递问题也不会在对话框中出现。Windows 会处理传递对话框中的控制之间的输入焦点所需的全部工作。

不过，在程序中添加对话框要比添加图标或菜单更麻烦一些。我们将从一个简单的对话框开始，让您对各部分之间的相互联系有所了解。

11.1 模态对话框

对话框分两类:"模态的"和"非模态的"。模态对话框最为普遍。当您的程序显示一个模态对话框时,用户不能在对话框与同一个程序中的另一个窗口之间进行切换,用户必须显式地结束该对话框,这通常通过按一下标记为 OK 或者 Cancel 的下压按钮来完成。不过,在显示模态对话框时,用户通常可以从当前程序切换到其他程序。有些对话框(称为"系统模态")甚至连这样的操作也不允许。在 Windows 中,显示了系统模态对话框之后,要完成其他任何工作,必须先结束该对话框。

11.1.1 创建 About 对话框

Windows 程序即使不需要接收用户输入,通常也具有由菜单上的 About 选项激活的对话框,该对话框用来显示程序的名称、图标、版权标志,以及标记为 OK 的下压按钮,也许还会有其他信息(例如技术支持的电话号码等)。我们将要看到的第一个程序除了显示一个 About 对话框外,别无它用。这个 ABOUT1 程序如程序 11-1 所示。

程序 11-1　ABOUT1 程序

ABOUT1.C

```
/*----------------------------------------
     ABOUT1.C——About Box Demo Program No. 1
               (c) Charles Petzold, 1998
  ----------------------------------------*/

#include <windows.h>
#include "resource.h"

LRESULT CALLBACK WndProc      (HWND, UINT, WPARAM, LPARAM) ;
BOOL    CALLBACK AboutDlgProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("About1") ;
     MSG          msg ;
     HWND         hwnd ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
```

```
	wndclass.cbWndExtra     = 0 ;
	wndclass.hInstance      = hInstance ;
	wndclass.hIcon          = LoadIcon (hInstance, szAppName) ;
	wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
	wndclass.hbrBackground  = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
	wndclass.lpszMenuName   = szAppName ;
	wndclass.lpszClassName  = szAppName ;

	if (!RegisterClass (&wndclass))
	{
		MessageBox (NULL, TEXT ("This program requires Windows NT!"),
		            szAppName, MB_ICONERROR) ;
		return 0 ;
	}

	hwnd = CreateWindow (szAppName, TEXT ("About Box Demo Program"),
	                     WS_OVERLAPPEDWINDOW,
	                     CW_USEDEFAULT, CW_USEDEFAULT,
	                     CW_USEDEFAULT, CW_USEDEFAULT,
	                     NULL, NULL, hInstance, NULL) ;

	ShowWindow (hwnd, iCmdShow) ;
	UpdateWindow (hwnd) ;

	while (GetMessage (&msg, NULL, 0, 0))
	{
		TranslateMessage (&msg) ;
		DispatchMessage (&msg) ;
	}
	return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
	static HINSTANCE hInstance ;
	switch (message)
	{
	case WM_CREATE :
		hInstance = ((LPCREATESTRUCT) lParam)->hInstance ;
		return 0 ;

	case WM_COMMAND :
		switch (LOWORD (wParam))
		{
		case IDM_APP_ABOUT :
			DialogBox (hInstance, TEXT ("AboutBox"), hwnd, AboutDlgProc) ;
			break ;
```

```
          }
          return 0 ;

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}

BOOL CALLBACK AboutDlgProc (HWND hDlg, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
     switch (message)
     {
     case WM_INITDIALOG :
          return TRUE ;

     case WM_COMMAND :
          switch (LOWORD (wParam))
          {
          case IDOK :
          case IDCANCEL :
               EndDialog (hDlg, 0) ;
               return TRUE ;
          }
          break ;
     }
     return FALSE ;
}
```

ABOUT1.RC (摘录)

```
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
// Dialog

ABOUTBOX DIALOG DISCARDABLE 32, 32, 180, 100
STYLE DS_MODALFRAME | WS_POPUP
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON   "OK",IDOK,66,80,50,14
    ICON            "ABOUT1",IDC_STATIC,7,7,21,20
    CTEXT           "About1",IDC_STATIC,40,12,100,8
```

```
    CTEXT           "About Box Demo Program",IDC_STATIC,7,40,166,8
    CTEXT           "(c) Charles Petzold, 1998",IDC_STATIC,7,52,166,8
END

/////////////////////////////////////////////////////////////////////////////
// Menu

ABOUT1 MENU DISCARDABLE
BEGIN
    POPUP "&Help"
    BEGIN
        MENUITEM "&About About1...",            IDM_APP_ABOUT
    END
END

/////////////////////////////////////////////////////////////////////////////
// Icon

ABOUT1                  ICON    DISCARDABLE     "About1.ico"
```

RESOURCE.H（摘录）

```
// Microsoft Developer Studio generated include file.
// Used by About1.rc

#define IDM_APP_ABOUT                   40001
#define IDC_STATIC                      -1
```

ABOUT1.ICO

按后面章节中介绍的方法,您还可以在程序中创建图标和菜单。图标和菜单的 ID 名均为“About1”。菜单有一个选项,它产生一个 ID 为 IDM_APP_ABOUT 的 WM_COMMAND 消息。这使程序显示图 11-1 所示的对话框。

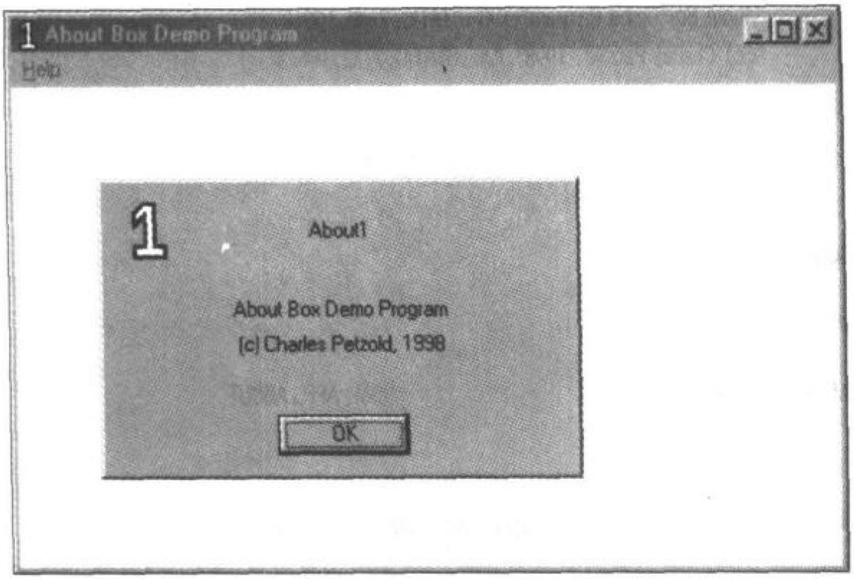


图 11-1 程序 ABOUT1 的对话框

11.1.2 对话框及其模板

为把一个对话框添加到 Visual C++ Developer Studio 中的应用程序上，可以先从 Insert 菜单中选择 Resource，然后选择 Dialog Box。现在一个对话框就出现在您的眼前，该对话框带有标题栏、标题（Dialog)，以及 OK 和 Cancel 按钮。Controls 工具栏允许您在对话框中插入不同的控件。

Developer Studio 将对话框的 ID 设为标准的 IDD_DIALOG1。您可以在此名称上（或者在对话框本身上）右键单击，然后从菜单中选择 Properties。在本程序中，将 ID 改为 AboutBox(带引号)。为了与我创建的对话框保持一致，请将 X Pos 和 Y Pos 域改为 32。这表示对话框相对于程序窗口客户区左上角的显示位置。（我马上就会详细地讨论对话框的坐标。）

现在，还是在 Properties 对话框中选择 Styles 选项卡。因为此对话框没有标题栏，所以不要选中 Title Bar 复选框。单击 Properties 对话框的关闭按钮。

现在可以设计对话框了。因为不需要 Cancel 按钮，所以单击该按钮，然后按键盘上的 Delete 键。单击 OK 按钮，然后将其移动到对话框的底部。在 Developer Studio 窗口下面的工具栏上有一个小位图，它使控件在窗口内水平居中对齐，按下此按钮。

如果您需要程序的图标出现在对话框中，可以这样做：在浮动的 Controls 工具栏中按下 Pictures 按钮。将鼠标移动到对话框的表面，按下左键，然后拖出一个矩形。这就是图标将出现的位置。在此矩形上按下鼠标右键，然后从菜单中选择 Properties。保持 ID 为 IDC_STATIC。此标识符在 RESOURCE.H 中定义为 -1，用于 C 程序不涉及的所有

ID。将 Type 改为 Icon。您可以在 Image 域输入程序图标的名称，或者，如果您已经创建了一个图标，那么您也可以从组合框中选择一个名称(About1)。

对于对话框中的三个静态文本串，可以从 Controls 工具栏选择 Static Text，然后确定文本在对话框中的位置。右键单击控件，然后从菜单中选择 Properties。在 Properties 框的 Caption 域中输入要显示的文本。选择 Styles 选项卡，然后从 Align Text 域选择 Center。

在添加这些文本串的时候，您可能希望对话框更大一些，请选中对话框，然后拖动边框。您也可以选择并缩放控件。通常用键盘上的光标移动键完成此操作会更容易些。箭头键本身移动控件，按下 Shift 键后按箭头键，可以改变控件的大小。所选控件的坐标和大小显示在 Developer Studio 窗口的右下角。

如果您建立了一个应用程序，那么以后在查看资源描述文件 ABOUT1.RC 时，您将发现 Developer Studio 建立的模板。我所设计的对话框的模板如下：

```
ABOUTBOX DIALOG DISCARDABLE  32, 32, 180, 100
STYLE DS_MODALFRAME | WS_POPUP
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON       "OK",IDOK,66,80,50,14
    ICON                "ABOUT1",IDC_STATIC,7,7,21,20
    CTEXT               "About1",IDC_STATIC,40,12,100,8
    CTEXT               "About Box Demo Program",IDC_STATIC,7,40,166,8
    CTEXT               "(c) Charles Petzold, 1998",IDC_STATIC,7,52,166,8
END
```

第一行给出了对话框的名称(这里为 ABOUTBOX)。由于其他原因，您也可以使用数字。名称后面是关键字 DIALOG 和 DISCARDABLE 以及 4 个数字。前两个数字是对话框左上角的 x、y 坐标，该坐标是程序调用对话框时相对于父窗口客户区的位置。后两个数字是对话框的宽度和高度。

这些坐标和大小的单位都不是像素。它们实际上是基于一种特殊的坐标系统，该系统只用于对话框模板。数字基于对话框中使用的字体的大小(这里是 8 磅的 MS Sans Serif 字体)：x 坐标，以及宽度是以字符平均宽度的 1/4 来表示；y 坐标，以及高度是以字符高度的 1/8 来表示。因此，对于这个对话框来说，对话框的左上角距离主窗口客户区的左边是 5 个字符，距离主窗口客户区的顶边是 $2\frac{1}{2}$ 个字符。对话框本身宽 40 个字符，高 10 个字符。

这样的坐标系使得程序员可以使用坐标和大小来大致勾勒对话框的尺寸和外观，而不管视频显示器的分辨率是多少。由于系统字体字符的高度大致为其宽度的两倍，所以，x 轴和 y 轴的量度差不多相等。

模板中的 STYLE 语句类似于 CreateWindow 调用中的 style 字段。对于模态对话框，

通常使用 WS_POPUP 和 DS_MODALFRAME，我们将在以后介绍一些其他的选项。

在 BEGIN 和 END 语句（或者是左右花括号，手工设计对话框模板时，您可能会使用）之间，定义出现在对话框中的子窗口控制。这个对话框使用了三种类型的子窗口控制，它们分别是 DEFPUSHBUTTON（默认下压按钮）、ICON（图标）和 CTEXT（文本居中）。这些语句的格式为

```
control-type "text" id, xPos, yPos, xWidth, yHeight, iStyle
```

其中，后面的 iStyle 项是可选的，它使用 Windows 头文件中定义的标识符来指定其他窗口风格。

DEFPUSHBUTTON、ICON 和 CTEXT 等标识符只可以在对话框中使用，它们是某种特定窗口类和窗口风格的缩写。例如，CTEXT 指示这个子窗口控制类是“静态的”，其风格为

```
WS_CHILD | SS_CENTER | WS_VISIBLE | WS_GROUP
```

虽然前面没有出现过 WS_GROUP 标识符，但是我们在第九章的 COLORS1 程序中已经出现过 WS_CHILD、SS_CENTER 和 WS_VISIBLE 窗口风格外，在创建静态子窗口文本控制时已经用到了它们。

对于图标，文本字段是程序的图标资源名称，它也在 ABOUT1 资源描述文件中定义。对于下压按钮，文本字段是出现在下压按钮里的文本，这个文本等价于在程序中创建子窗口控制时调用 CreateWindow 所指定的第二个参数。

id 字段是子窗口在向其父窗口发送消息（通常为 WM_COMMMAND 消息）时用来标识它自身的值。这些子窗口控制的父窗口就是对话柜本身，它将这些消息发送给 Windows 的一个窗口过程。不过，这个窗口过程也将这些消息发送给在程序中给出的对话框过程。ID 值等价于我们在第九章创建子窗口时，在 CreateWindow 函数中使用的子窗口 ID。由于文本和图标控制不向父窗口回送消息，所以这些值被设置为 IDC_STATIC，它在 RESOURCE.H 中定义为 -1。下压按钮的 ID 值为 IDOK，它在 WINUSER.H 中定义为 1。

接下去的 4 个数设置子窗口的位置（相对于对话框客户区的左上角）和大小，它们是以系统字体平均宽度的 1/4 和平均高度的 1/8 为单位来表示的。对于 ICON 语句，宽度和高度将被忽略。

对话框模板中的 DEFPUSHBUTTON 语句，除了包含 DEFPUSHBUTTON 关键字所隐含的窗口风格外，还包含窗口风格 WS_GROUP。在后面讨论该程序的第二个版本 ABOUT2 时，我还要在后面详细地介绍有关 WS_GROUP（以及相关的 WS_TABSTOP 风格）的情况。

11.1.3 对话框过程

程序内的对话框过程处理传给对话框的消息。尽管看起来很像是窗口过程,但是它并不是真实的窗口过程。对话框的窗口过程在 Windows 内部定义,这个窗口过程调用您编写的对话框过程,把它所接收到的许多消息作为参数。下面是 ABOUT1 的对话框过程:

```
BOOL CALLBACK AboutDlgProc (HWND hDlg, UINT message,
                           WPARAM wParam, LPARAM lParam)
{
     switch (message)
     {
     case WM_INITDIALOG :
          return TRUE ;

     case WM_COMMAND :
          switch (LOWORD (wParam))
          {
          case IDOK :
          case IDCANCEL :
               EndDialog (hDlg, 0) ;
               return TRUE ;
          }
          break ;
     }
     return FALSE ;
}
```

该函数的参数与常规窗口过程的参数相同;与窗口过程类似,对话框过程必须定义为一个 CALLBACK(回调)函数。尽管我使用了 hDlg 作为对话框窗口的句柄,但是您也可以按照您自己的意愿使用 hwnd。首先,让我们来看一下这个函数与窗口过程的区别:

- 窗口过程返回一个 LRESULT;对话框返回一个 BOOL,它在 Windows 头文件中定义为 int 类型;
- 如果窗口过程不处理某个特定的消息,那么它将调用 DefWindowProc;如果对话框过程处理一个消息,那么它返回 TRUE(非 0),如果不处理,则返回 FALSE(0);
- 对话框过程不需要处理 WM_PAINT 或 WM_DESTROY 消息。对话框过程不接收 WM_CREAT 消息;而是在特殊的 WM_INITDIALOG 消息期间,对话框过程执行初始化操作。

WM_INITDIALOG 消息是对话框接收到的第一个消息,这个消息只发送给对话框过

程。如果对话框过程返回 TRUE,那么 Windows 将输入焦点设置给对话框中第一个具有 WS_TABSTOP 风格(我们将在 ABOUT2 的讨论中加以解释)的子窗口控制。在这个对话框中,第一个具有 WS_TABSTOP 风格的子窗口控制是下压按钮。另外,对话框过程也可以在处理 WM_INITDIALOG 时使用 SetFocus 来将输入焦点设置为对话框中的某个子窗口控制,然后返回 FALSE。

此外,对话框过程只处理 WM_COMMAND 消息。这是当下压按钮被鼠标点中,或者如果在按钮具有输入焦点的情况下按下 Spacebar 键时,下压按钮控制发送给其父窗口的消息。这个控制的 ID(我们在对话框模板中将其设置为 IDOK)在 wParam 的低位字中,对于这个消息,对话框过程调用 EndDialog,它告诉 Windows 清除对话框。对于所有其他消息,对话框过程返回 FALSE,并告诉 Windows 内部的对话框窗口过程:我们的对话框过程不处理这些消息。

模态对话框的消息不通过程序的消息队列,所以不必担心对话框中键盘加速键的影响。

11.1.4 激活对话框

在 WndProc 中处理 WM_CREATE 消息时,ABOUT1 获取程序的实例句柄,并将它放在静态变量中:

```
hInstance = ((LPCREATESTRUCT) lParam)->hInstance ;
```

ABOUT1 检查 WM_COMMAND 消息,以确保消息 wParam 的低位字等于 IDM_APP_ABOUT。当它获得这样一个消息时,程序调用 DialogBox:

```
DialogBox (hInstance, TEXT ("AboutBox"), hwnd, AboutDlgProc) ;
```

该函数需要实例句柄(在处理 WM_CREATE 时保存的)、对话框名称(在资源描述文件中定义的)、对话框的父窗口(也是程序的主窗口),以及对话框过程的地址。如果使用一个数字而不是对话框模板名称,那么可以用 MAKEINTRESOURCE 宏将它转换为一个串。

从菜单中选择 About About1 将显示图 11-1 所示的对话框。您可以用鼠标单击 OK 按钮、按 Spacebar 键,或者按 Enter 键来结束这个对话框。对于任意包含默认按钮的对话框,在按 Enter 键或 Spacebar 键之后,Windows 给对话框发送一个 WM_COMMAND 消息,并令 wParam 的低位字等于默认下压按钮的 ID,这时的 ID 为 IDOK。按下 Escape 键也可以关闭对话框,这时 Windows 将发送一个 WM_COMMAND 消息,并令 ID 等于 IDCANCEL。

直到对话框结束之后,用来显示对话框的 DialogBox 才将控制返回给 WndProc。DialogBox 的返回值是对话框过程内部调用的 EndDialog 函数的第二个参数(这个值未在

ABOUT1 中使用,但是在 ABOUT2 中使用了)。然后,WndProc 可以将控制返回给 Windows。

即使在显示对话框时,WndProc 也可以继续接收消息。实际上,您可以从对话框过程内部给 WndProc 发送消息。ABOUT1 的主窗口是弹出式对话框窗口的父窗口,所以 AboutDlgProc 中的 SendMessage 调用能以语句

```
SendMessage (GetParent (hDlg), . . . ) ;
```

开始。

11.1.5 不同的主题

虽然 Visual C++ Developer Studio 中的对话框编辑器和其他资源编辑器使我们几乎不用考虑资源描述,但是学习一些资源描述的语法还是有用的。尤其对于对话框模板,知道了语法,您就可以更多地了解对话框的范围和限制。甚至当它不能满足您的需要时,您还可以自己建立一个对话框模板(就像本章后面的 HEXCALC 程序)。资源编译器和资源描述语法的文档位于/Platform SDK/Windows Programming Guidelines/Platform SDK Tools/Compiling/Using the Resource Compiler。

在 Developer Studio 的 Properties 对话框中指定了对话框的窗口风格,它翻译成对话框模板中的 STYLE 语句。对于 ABOUT1,我们使用模态对话框最常用的风格;

```
STYLE WS_POPUP | DS_MODALFRAME
```

然而,您也可以尝试其他风格。有些对话框有标题栏,标题栏用于指出对话框的用途,并允许用户通过鼠标在显示屏上移动对话框。此风格为 WS_CAPTION。如果您使用 WS_CAPTION,那么 DIALOG 语句中所指定的 x 坐标和 y 坐标是对话框客户区的坐标,并相对于父窗口客户区的左上角。标题栏将在 y 坐标之上显示。

如果使用了标题栏,那么您可以用 CAPTION 语句将文本放入标题中。在对话框模板中,CAPTION 语句在 STYLE 语句的后面:

```
CAPTION "Dialog Box Caption"
```

另外,在对话框过程处理 WM_INITDIALOG 消息期间,您还可以调用:

```
SetWindowText (hDlg, TEXT ("Dialog Box Caption")) ;
```

如果您使用 WS_CAPTION 风格,也可以添加一个 WS_SYSMENU 风格的系统菜单框。此风格允许用户从系统菜单中选择 Move 或 Close。

从 Properties 对话框的 Border 列表框中选择 Resizing(等价于风格 WS_THICKFRAME),允许用户缩放对话框,尽管此操作不常用。如果您不介意更特殊一点的话,还

可以试一下给此对话框风格添加最大化框。

您甚至可以给对话框添加一个菜单。这时对话框模板将包括语句

```
MENU menu-name
```

其参数既是菜单的名称，又是资源描述中的菜单号。模态对话框很少使用菜单。如果使用了菜单，那么您必须确保菜单和对话框控制中的所有 ID 号都是唯一的，或者不唯一，但它们却表达了相同的命令。

FONT 语句使您可以设置非系统字体，以供对话框文本使用。过去这在对话框中不常用，但现在却非常普遍。事实上，默认情况下，Developer Studio 为您创建的每一个对话框都选用 8 磅的 MS Sans Serif 字体。一个 Windows 程序能够完成查找工作，这只需利用程序为对话框及其他文本输出而单独使用的某种字体即可。

尽管对话框窗口过程通常位于 Windows 内部，但是您也可以使用自己编写的窗口过程来处理对话框消息。要这样做，您必须在对话框模板中指定一个窗口类名：

```
CLASS "class-name"
```

这种用法很少见，但是在本章后面所示的 HEXCALC 程序中我们将用到它。

当您使用对话框模板的名称来调用 DialogBox 时，Windows 通过调用普通的 CreateWindow 函数来完成创建弹出式窗口所需要完成的一切操作。Windows 从对话框模板中获取窗口的坐标、大小、窗口风格、标题和菜单，从 DialogBox 的参数中获取实例句柄和父窗口句柄。它所需要的唯一的其他信息是一个窗口类(假设对话框模板不指定窗口类的话)。Windows 为对话框注册一个专用的窗口类，这个窗口类的窗口过程可以访问对话框过程地址(该地址是您在 DialogBox 调用中指定的)，所以它可以使程序获得该弹出式窗口所接收的消息。当然，您可以通过自己创建弹出式窗口来创建和维护自己的对话框。不过，使用 DialogBox 则更简单。

虽然您希望受益于 Windows 对话框管理器，但不希望(或者能够)在资源描述中定义对话框模板，也可能您希望程序在运行时可以动态地创建对话框。完成此功能的函数是 DialogBoxIndirect，此函数用数据结构来定义模板。

在 ABOUT1.RC 的对话框模板中，我们使用缩写 CTEXT、ICON 和 DEFPUSHBUTTON 来定义对话框所需要的三种类型的子窗口控制。您还可以使用其他类型，每种类型都隐含一个特定的预先定义的窗口类和一种窗口风格。下页表 11-1 显示了每种控制类型的等价窗口类和窗口风格。

资源编译器是唯一能够识别这些缩写的程序。除了表中所示的窗口风格外，每个控制还具有下面的风格：

```
WS_CHILD | WS_VISIBLE
```

对于这些控制类型，除了 EDITTEXT、SCROLLBAR、LISTBOX 和 COMBOBOX 外，控制

语句的格式为

```
control-type "text", id, xPos, yPos, xWidth, yHeight, iStyle
```

表 11-1

控制类型	窗口类	窗口风格
PUSHBUTTON	按钮	BS_PUSHBUTTON | WS_TABSTOP
DEFPUSHBUTTON	按钮	BS_DEFPUSHBUTTON | WS_TABSTOP
CHECKBOX	按钮	BS_CHECKBOX | WS_TABSTOP
RADIOBUTTON	按钮	BS_RADIOBUTTON | WS_TABSTOP
GROUPBOX	按钮	BS_GROUPBOX | WS_TABSTOP
LTEXT	静态	SS_LEFT | WS_GROUP
CTEXT	静态	SS_CENTER | WS_GROUP
RTEXT	静态	SS_RIGHT | WS_GROUP
ICON	静态	SS_ICON
EDITTEXT	编辑	ES_LEFT | WS_BORDER | WS_TABSTOP
SCROLLBAR	滚动条	SBS_HORZ
LISTBOX	列表框	LBS_NOTIFY | WS_BORDER | WS_VSCROLL
COMBOBOX	组合框	CBS_SIMPLE | WS_TABSTOP

对于 EDITTEXT、SCROLLBAR、LISTBOX 和 COMBOBOX，其格式为

```
control-type id, xPos, yPos, xWidth, yHeight, iStyle
```

其中没有文本字段。在这两种语句中，iStyle 参数都是可选的。

在第九章，我讨论了确定预先定义的子窗口的宽度和高度的规则。您可能需要回到第九章去参考这些规则，这时请记住：对话框模板中指定大小的单位为平均字符宽度的 1/4，平均字符高度的 1/8。

控制语句的 style 字段是可选的。它允许您包含其他窗口风格标识符。例如，如果您想创建在正方形框左边包含文本的复选框，那么可以使用

```
CHECKBOX "text", id, xPos, yPos, xWidth, yHeight, BS_LEFTTEXT
```

注意，控制类型 EDITTEXT 会自动添加一个边框。如果您想创建一个没有边框的子窗口编辑控制，您可以使用

```
EDITTEXT id, xPos, yPos, xWidth, yHeight, NOT WS_BORDER
```

资源编译器也承认与下面语句类似的专用控制语句：

```
CONTROL "text", id, "class", iStyle, xPos, yPos, xWidth, yHeight
```

此语句允许您通过指定窗口类和完整的窗口风格来创建任意类型的子窗口控制。例如，要取代

```
PUSHBUTTON "OK", IDOK, 10, 20, 32, 14
```

您可以使用：

```
CONTROL "OK", IDOK, "button", WS_CHILD | WS_VISIBLE |
        BS_PUSHBUTTON | WS_TABSTOP, 10, 20, 32, 14
```

当编译资源描述文件时，这两条语句在.RES和.EXE文件中的编码是相同的。在Developer Studio中，您可以使用Controls工具栏中的Custom Control选项来创建此语句。在程序11-3所示的ABOUT3中，我向您展示了如何用此选项创建一个控制，且在您的程序中已定义了该控制的窗口类。

当您在对话框模板中使用CONTROL语句时，不必包含WS_CHILD和WS_VISIBLE风格。在创建子窗口时，Windows包含了这些窗口风格。CONTROL语句的格式也说明Windows对话框管理器在创建对话框时就完成了此操作。首先，就像前面所讨论的，它创建一个弹出式窗口，其父窗口句柄在DialogBox函数中提供。然后，对话框管理器为对话框模板中的每个控制创建一个子窗口。所有这些控制的父窗口均是这个弹出式对话框。上面给出的CONTROL语句被转换成一个CreateWindow调用，形式如下所示：

```
hCtrl = CreateWindow (TEXT ("button"), TEXT ("OK"),
                      WS_CHILD | WS_VISIBLE | WS_TABSTOP | BS_PUSHBUTTON,
                      10 * cxChar / 4, 20 * cyChar / 8,
                      32 * cxChar / 4, 14 * cyChar / 8,
                      hDlg, IDOK, hInstance, NULL) ;
```

其中，cxChar和cyChar是系统字体字符的宽度和高度，以像素为单位；hDlg参数是从创建该对话框窗口的CreateWindow调用返回的值；hInstance参数是从DialogBox调用获得的。

11.1.6 更复杂的对话框

ABOUT1中的简单对话框演示了设计和运行一个对话框的要点；现在让我们来看一个稍微复杂些的例子。程序11-2给出的ABOUT2程序演示了如何在对话框过程中管理控制(这里用单选按钮)，以及如何在对话框的客户区中绘图。

程序11-2 ABOUT2程序

ABOUT2.C

```
/*----------------------------------------
   ABOUT2.C——About Box Demo Program No. 2
             (c) Charles Petzold, 1998
  ----------------------------------------*/
```

```
#include <windows.h>
#include "resource.h"

LRESULT CALLBACK WndProc      (HWND, UINT, WPARAM, LPARAM) ;
BOOL    CALLBACK AboutDlgProc (HWND, UINT, WPARAM, LPARAM) ;

int iCurrentColor  = IDC_BLACK,
    iCurrentFigure = IDC_RECT ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("About2") ;
     MSG          msg ;
     HWND         hwnd ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (hInstance, szAppName) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = szAppName ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("About Box Demo Program"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
```

```
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

void PaintWindow (HWND hwnd, int iColor, int iFigure)
{
     static COLORREF crColor[8] = { RGB (  0,   0, 0), RGB (  0,   0, 255),
                                    RGB (  0, 255, 0), RGB (  0, 255, 255),
                                    RGB (255,   0, 0), RGB (255,   0, 255),
                                    RGB (255, 255, 0), RGB (255, 255, 255) } ;

     HBRUSH          hBrush ;
     HDC             hdc ;
     RECT            rect ;

     hdc = GetDC (hwnd) ;
     GetClientRect (hwnd, &rect) ;
     hBrush = CreateSolidBrush (crColor[iColor-IDC_BLACK]) ;
     hBrush = (HBRUSH) SelectObject (hdc, hBrush) ;

     if (iFigure == IDC_RECT)
          Rectangle (hdc, rect.left, rect.top, rect.right, rect.bottom) ;
     else
          Ellipse (hdc, rect.left, rect.top, rect.right, rect.bottom) ;

     DeleteObject (SelectObject (hdc, hBrush)) ;
     ReleaseDC (hwnd, hdc) ;
}

void PaintTheBlock (HWND hCtrl, int iColor, int iFigure)
{
     InvalidateRect (hCtrl, NULL, TRUE) ;
     UpdateWindow (hCtrl) ;
     PaintWindow (hCtrl, iColor, iFigure) ;
}
LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static HINSTANCE  hInstance ;
     PAINTSTRUCT       ps ;

     switch (message)
     {
     case WM_CREATE:
          hInstance = ((LPCREATESTRUCT) lParam) -> hInstance ;
          return 0 ;

     case WM_COMMAND:
```

```
          switch (LOWORD (wParam))
          {
          case IDM_APP_ABOUT:
               if (DialogBox (hInstance, TEXT ("AboutBox"), hwnd, AboutDlgProc))
                    InvalidateRect (hwnd, NULL, TRUE) ;
               return 0 ;
          }
          break ;

     case WM_PAINT:
          BeginPaint (hwnd, &ps) ;
          EndPaint (hwnd, &ps) ;

          PaintWindow (hwnd, iCurrentColor, iCurrentFigure) ;
          return 0 ;

     case WM_DESTROY:
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}

BOOL CALLBACK AboutDlgProc (HWND hDlg, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
     static HWND hCtrlBlock ;
     static int  iColor, iFigure ;

     switch (message)
     {
     case WM_INITDIALOG:
          iColor  = iCurrentColor ;
          iFigure = iCurrentFigure ;

          CheckRadioButton (hDlg, IDC_BLACK, IDC_WHITE,   iColor) ;
          CheckRadioButton (hDlg, IDC_RECT,  IDC_ELLIPSE, iFigure) ;

          hCtrlBlock = GetDlgItem (hDlg, IDC_PAINT) ;

          SetFocus (GetDlgItem (hDlg, iColor)) ;
          return FALSE ;

     case WM_COMMAND:
          switch (LOWORD (wParam))
          {
          case IDOK:
```

```
                    iCurrentColor = iColor ;
                    iCurrentFigure = iFigure ;
                    EndDialog (hDlg, TRUE) ;
                    return TRUE ;

               case IDCANCEL:
                    EndDialog (hDlg, FALSE) ;
                    return TRUE ;

               case IDC_BLACK:
               case IDC_RED:
               case IDC_GREEN:
               case IDC_YELLOW:
               case IDC_BLUE:
               case IDC_MAGENTA:
               case IDC_CYAN:
               case IDC_WHITE:
                    iColor = LOWORD (wParam) ;
                    CheckRadioButton (hDlg, IDC_BLACK, IDC_WHITE, LOWORD (wParam)) ;
                    PaintTheBlock (hCtrlBlock, iColor, iFigure) ;
                    return TRUE ;

               case IDC_RECT:
               case IDC_ELLIPSE:
                    iFigure = LOWORD (wParam) ;
                    CheckRadioButton (hDlg, IDC_RECT, IDC_ELLIPSE, LOWORD (wParam)) ;
                    PaintTheBlock (hCtrlBlock, iColor, iFigure) ;
                    return TRUE ;
               }
               break ;

          case WM_PAINT:
               PaintTheBlock (hCtrlBlock, iColor, iFigure) ;
               break ;
          }
          return FALSE ;
}
```

ABOUT2.RC（摘录）

```
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
// Dialog
```

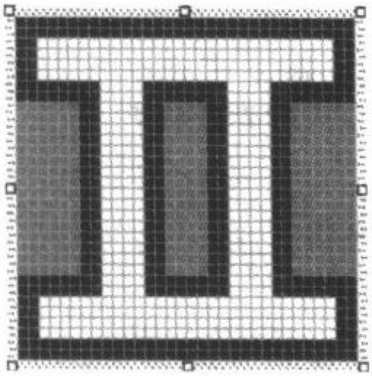
```
ABOUTBOX DIALOG DISCARDABLE 32, 32, 200, 234
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION
FONT 8, "MS Sans Serif"
BEGIN
    ICON            "ABOUT2",IDC_STATIC,7,7,20,20
    CTEXT           "About2",IDC_STATIC,57,12,86,8
    CTEXT           "About Box Demo Program",IDC_STATIC,7,40,186,8
    LTEXT           "",IDC_PAINT,114,67,74,72
    GROUPBOX        "&Color",IDC_STATIC,7,60,84,143
    RADIOBUTTON     "&Black",IDC_BLACK,16,76,64,8,WS_GROUP | WS_TABSTOP
    RADIOBUTTON     "B&lue",IDC_BLUE,16,92,64,8
    RADIOBUTTON     "&Green",IDC_GREEN,16,108,64,8
    RADIOBUTTON     "Cya&n",IDC_CYAN,16,124,64,8
    RADIOBUTTON     "&Red",IDC_RED,16,140,64,8
    RADIOBUTTON     "&Magenta",IDC_MAGENTA,16,156,64,8
    RADIOBUTTON     "&Yellow",IDC_YELLOW,16,172,64,8
    RADIOBUTTON     "&White",IDC_WHITE,16,188,64,8
    GROUPBOX        "&Figure",IDC_STATIC,109,156,84,46,WS_GROUP
    RADIOBUTTON     "Rec&tangle",IDC_RECT,116,172,65,8,WS_GROUP | WS_TABSTOP
    RADIOBUTTON     "&Ellipse",IDC_ELLIPSE,116,188,64,8
    DEFPUSHBUTTON   "OK",IDOK,35,212,50,14,WS_GROUP
    PUSHBUTTON      "Cancel",IDCANCEL,113,212,50,14,WS_GROUP
END

/////////////////////////////////////////////////////////////////////////////
// Icon

ABOUT2                  ICON    DISCARDABLE     "About2.ico"

/////////////////////////////////////////////////////////////////////////////
// Menu

ABOUT2 MENU DISCARDABLE
BEGIN
    POPUP "&Help"
    BEGIN
        MENUITEM "&About",                      IDM_APP_ABOUT
    END
END
```

RESOURCE.H (摘录)

```
// Microsoft Developer Studio generated include file.
// Used by About2.rc

#define IDC_BLACK                       1000
```

```
#define IDC_BLUE                1001
#define IDC_GREEN               1002
#define IDC_CYAN                1003
#define IDC_RED                 1004
#define IDC_MAGENTA             1005
#define IDC_YELLOW              1006
#define IDC_WHITE               1007
#define IDC_RECT                1008
#define IDC_ELLIPSE             1009
#define IDC_PAINT               1010
#define IDM_APP_ABOUT          40001
#define IDC_STATIC                -1
```

ABOUT2.ICO



ABOUT2 中的 About 框有两组单选按钮。一组用来选择颜色,另一组用来选择是矩形还是椭圆。所选的矩形或者椭圆显示在对话框内,其内部以当前选定的颜色着色。用户按下 OK 按钮后,对话框终止,程序的窗口过程在它自己的客户区内绘出所选图形。如果按下 Cancel,则主窗口的客户区保持原样。对话框如下页图 11-2 所示。尽管 ABOUT2 使用预先定义的标识符 IDOK 和 IDCANCEL 作为两个下压按钮,但是每个单选按钮均有自己的标识符,它们以字母 IDC 打头(ID 用于控制)。这些标识符在 RESOURCE.H 中定义。

当您在 ABOUT2 对话框中创建单选按钮时,请按显示顺序创建。这能保证 Developer Studio 顺序定义标识符的值,程序将使用这些值。另外,每个单选按钮都不要选中 Auto 选项。Auto Radio Button 需要较少的代码,但开始时会有些深奥。按 ABOUT2.RC 设置它们的标识符。

选中 Properties 对话框中下列对象的 Group 选项:OK 和 Cancel 按钮、Figure 组合框、每个组合框中的第一个单选按钮(Black 和 Rectangle)。选中这两个单选按钮的 Tab Stop 复选框。

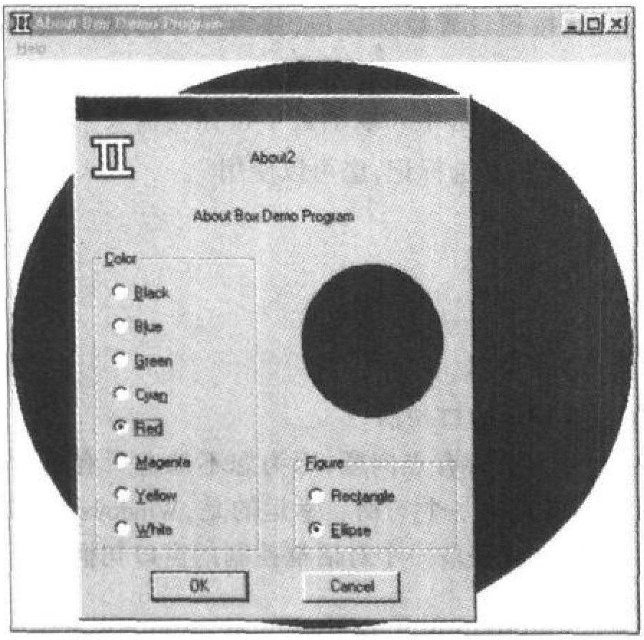


图 11-2　ABOUT2 程序的对话框

当您有全部控件在对话框中的近似位置和大小时，就可以从 Layout 菜单选择 Tab Order 选项。按 ABOUT2.RC 资源描述中显示的顺序单击每一个控件。

11.1.7　使用对话框控制

在第九章中，您发现大多数子窗口控制给其父窗口发送 WM_COMMAND 消息（唯一例外的是滚动条控制）。您还看到，通过给子窗口控制发消息，父窗口可以改变子窗口控制（例如，选中或者取消选中单选按钮或者复选框）。您也可以类似地在对话框过程中改变控制。例如，如果您设计了一系列单选按钮，就可以通过给它们发送消息，用以选中或者取消选中这些按钮。不过，Windows 也提供了几种使用对话框控制的简捷方法。我们来看一看对话框过程与子窗口控制相互通信的方式。

ABOUT2 的对话框模板显示在程序 11-2 的 ABOUT2.RC 资源描述文件中。GROUP-BOX 控制只是一个带标题（标题为“Color”或者“Figure”）的框架，每组单选按钮都由这样的框架包围。前一组的 8 个单选按钮是互斥的，第二组的 2 个单选按钮也是如此。

当用鼠标单击其中一个单选按钮时（或者当单选按钮拥有输入焦点时按 Spacebar 键），子窗口向其父窗口发送一个 WM_COMMAND 消息，消息的 wParam 的低位字被置为控制的 ID，wParam 的高位字是一个通知码，lParam 值是控制的窗口句柄。对单选按钮，这个通知码是 BN_CLICKED 或者 0。然后，Windows 中的对话框窗口过程将这个 WM_COMMAND 消息发送给 ABOUT2.C 内的对话框过程。当对话框过程收到一个单选按钮

的 WM_COMMAND 消息时，它为此按钮设置复选标记，并为组中其他按钮清除复选标记。

您可能还记得在第九章中已经讲过，选中和取消选中按钮均需要向子窗口控制发送 BM_CHECK 消息。要设置一个按钮复选标记，您可以使用

```
SendMessage (hwndCtrl, BM_SETCHECK, 1, 0) ;
```

要消除复选标记，您可以使用

```
SendMessage (hwndCtrl, BM_SETCHECK, 0, 0) ;
```

其中 hwndCtrl 参数是子窗口按钮控制的窗口句柄。

但是在对话框过程中使用这种方法时有点问题，因为您不知道所有单选按钮的窗口句柄，只是从您获得的消息中知道其中一个句柄。幸运的是，Windows 为您提供了一个函数，可以用对话框句柄和控制 ID 来获取一个对话框控制的窗口句柄：

```
hwndCtrl = GetDlgItem (hDlg, id) ;
```

（您也可以使用如下函数从窗口句柄中获取控制的 ID 值：

```
id = GetWindowLong (hwndCtrl, GWL_ID) ;
```

但是在多数情况下这是不必要的。）

您会注意到，在程序 11-2 所示的头文件 ABOUT2.H 中，8 种颜色的 ID 值是从 IDC_BLACK 到 IDC_WHITE 连续变化的，这种安排在处理来自单选按钮的 WM_COMMAND 消息时将很有用。在第一次尝试选中或者取消选中单选按钮时，您可能会在对话框过程中编写如下的程序：

```
static int iColor ;
[other program lines]
case WM_COMMAND:
    switch (LOWORD (wParam))
    {
    [other program lines]
    case IDC_BLACK:
    case IDC_RED:
    case IDC_GREEN:
    case IDC_YELLOW:
    case IDC_BLUE:
    case IDC_MAGENTA:
    case IDC_CYAN:
    case IDC_WHITE:
        iColor = LOWORD (wParam) ;
```

```
    for (i = IDC_BLACK, i <= IDC_WHITE, i++)
        SendMessage (GetDlgItem (hDlg, i),
                    BM_SETCHECK, i == LOWORD (wParam), 0) ;
    return TRUE ;
[other program lines]
```

这种方法能令人满意地运行。将新的颜色值保存在 iColor 中，并且还建立了一个循环，轮流使用所有 8 种颜色的 ID 值。获取每个单选按钮控制的窗口句柄，并用 SendMessage 给每个句柄发送一个 BM_SETCHECK 消息。只有对于向对话框窗口过程发送 WM_COMMAND 消息的按钮，这个消息的 wParam 值才被设置为 1。

第一种简化的方法是使用专门的对话框过程 SendDlgItemMessage：

```
SendDlgItemMessage (hDlg, id, iMsg, wParam, lParam) ;
```

它等价于

```
SendMessage (GetDlgItem (hDlg, id), id, wParam, lParam) ;
```

现在，循环将如下所示：

```
for (i = IDC_BLACK, i <= IDC_WHITE, i++)
    SendDlgItemMessage (hDlg, i, BM_SETCHECK, i == LWORD (wParam), 0) ;
```

稍微有些改进。但是真正的重大突破要等到使用了 CheckRadioButton 函数时才会出现：

```
CheckRadioButton (hDlg, idFirst, idLast, idCheck) ;
```

这个函数将 ID 在 idFirst 到 idLast 之间的所有单选按钮控制的复选标记都清除掉，除了 ID 为 idCheck 的单选按钮，因为它是选中的。这里，所有 ID 必须是连续的。从而我们可以完全摆脱循环，并使用：

```
CheckRadioButton (hDlg, IDC_BLACK, IDC_WHITE, LOWORD (wParam)) ;
```

这正是 ABOUT2 对话框过程采用的方法。

在使用复选框时，也提供了类似的简化函数。如果您创建了一个“CHECKBOX”对话框窗口控制，那么可以使用如下函数来设置和清除复选标记：

```
CheckDlgButton (hDlg, idCheckbox, iCheck) ;
```

如果 iCheck 设置为 1，那么按钮被选中；如果设置为 0，那么按钮被取消选中。您可以使用如下方法来获取对话框中某个复选框的状态：

```
iCheck = IsDlgButtonChecked (hDlg, idCheckbox) ;
```

在对话框过程中，您既可以将复选标记的当前状态保存在一个静态变量中，又可以在收

到一个 WM_COMMAND 消息后，使用如下方法触发按钮：

```
CheckDlgButton (hDlg, idCheckbox,
    !IsDlgButtonChecked (hDlg, idCheckbox)) ;
```

如果您定义了 BS_AUTOCHECKBOX 控制，那么完全没有必要处理 WM_COMMAND 消息。在终止对话框前，您只要使用 IsDlgButtonChecked 就可以获取按钮的当前状态。不过，如果您使用 BS_AUTORADIOBUTTON 风格，那么 IsDlgButtonChecked 就不会令人满意，因为需要为每个单选按钮都调用它，直到函数返回 TRUE。实际上，您还要捕获 WM_COMMAND 消息来跟踪按下的按钮。

11.1.8 OK 和 Cancel 按钮

ABOUT2 有两个下压按钮，分别标记为 OK 和 Cancel。在 ABOUT2.RC 的对话框模板中，OK 按钮的 ID 值为 IDOK(在 WINUSER.H 中被定义为 1)，Cancel 按钮的 ID 值为 IDCANCEL(定义为 2)，OK 按钮是默认的：

```
DEFPUSHBUTTON     "OK",IDOK,35,212,50,14
PUSHBUTTON        "Cancel",IDCANCEL,113,212,50,14
```

在对话框中，通常都这样安排 OK 和 Cancel 按钮：将 OK 按钮作为默认按钮有助于用键盘界面终止对话。通常情况下，通过单击两个鼠标键之一或者当所期望的按钮具有输入焦点时按下 Spacebar 键来终止对话框。不过，如果用户按下 Enter 键，对话框窗口过程也将产生一个 WM_COMMAND 消息，而不管哪个控制拥有输入焦点。wParam 的低位字被设置为对话框中默认下压按钮的 ID 值，除非另一个下压按钮拥有输入焦点。在后一种情况下，wParam 的低位字被设置为拥有输入焦点的下压按钮的 ID 值。如果对话框中没有默认的下压按钮，那么 Windows 向对话框过程发送一个 WM_COMMAND 消息，消息中 wParam 的低位字被设置为 IDOK。如果用户按下 Esc 键或者 Ctrl-Break 键，那么 Windows 令 wParam 等于 IDCANCEL，并给对话框过程发送一个 WM_COMMAND 消息。所以，您不用在对话框过程中加入单独的处理键盘的操作，因为通常终止对话框的击键会由 Windows 将这两个下压按钮动作转换为 WM_COMMAND 消息。

AboutDlgProc 函数通过调用 EndDialog 来处理这两种 WM_COMMAND 消息：

```
switch (LWORD (wParam))
{
case IDOK:
    iCurrentColor = iColor ;
    iCurrentFigure = iFigure ;
    EndDialog (hDlg, TRUE) ;
    return TRUE ;
```

```
case IDCANCEL :
    EndDialog (hDlg, FALSE) ;
    return TRUE ;
```

ABOUT2 的窗口过程在程序的客户区中绘制矩形或椭圆时使用了全局变量 iCurrentColor 和 iCurrentFigure。AboutDlgProc 在对话框中画图时使用了静态局部变量 iColor 和 iFigure。

注意 EndDialog 的第二个参数的不同值,这个值是在 WndProc 中作为原 DialogBox 函数的返回值传回的:

```
case IDM_ABOUT:
    if (DialogBox (hInstance, TEXT ("AboutBox"), hwnd, AboutDlgProc))
        InvalidateRect (hwnd, NULL, TRUE) ;
    return 0 ;
```

如果 DialogBox 返回 TRUE(非 0),则意味着按下了 OK 按钮,然后需要使用新的颜色来刷新 WndProc 客户区。当 AboutDlgProc 收到一个 WM_COMMAND 消息,并且消息的 wParam 的低位字等于 IDOK 时,AboutDlgProc 将图形和颜色保存在全局变量 iCurrentColor 和 iCurrentFigure 中。如果 DialogBox 返回 FALSE,则主窗口继续使用 iCurrentColor 和 iCurrentFigure 的原始设置。

TRUE 和 FALSE 通常用于 EndDialog 调用中,以告知主窗口过程用户是用 OK 还是用 Cancel 来终止对话框的。不过,EndDialog 的参数实际上是一个 int 值,而 DialogBox 也返回一个 int 值,所以,用这种方法能比仅用 TRUE 或者 FALSE 返回更多的信息。

11.1.9 避免全局变量

在 ABOUT2 中使用全局变量可能会也可能不会影响您。一些程序员(包括我本人)都更喜欢尽量少地使用全局变量。ABOUT2 中的变量 iCurrentColor 和 iCurrentFigure 更像是全局定义的合法候选人,因为它们必须同时在窗口过程和对话框过程中使用。不过,一个具有很多对话框的程序,因其每个对话框能够改变若干个变量的值,故非常容易产生全局变量的急剧混乱。

您可能更倾向于认为程序中的对话框与数据结构相联系,该数据结构含有对话框可以改变的所有变量。您将在 typedef 语句中定义这些结构。例如,在 ABOUT2 中,可以定义与“About”框相联系的结构:

```
typedef struct
{
    int iColor, iFigure ;
}
ABOUTBOX_DATA ;
```

在 WndProc 中,您可以基于此结构来定义并初始化一个静态变量:

```
static ABOUTBOX_DATA ad = { IDC_BLACK, IDC_RECT } ;
```

在 WndProc 中也是这样,用 ad.iColor 和 ad.iFigure 替换所有的 iCurrentColor 和 iCurrentFigure 事件。调用对话框时,使用 DialogBoxParam 而不用 DialogBox。此函数的第 5 个参数可以是任意的 32 位值,一般地,此值设置为指向一个结构的指针,这里是 WndProc 中的 ABOUTBOX_DATA 结构:

```
case IDM_ABOUT:
    if (DialogBoxParam (hInstance, TEXT ("AboutBox"),
                        hwnd, AboutDlgProc, &ad))
        InvalidateRect (hwnd, NULL, TRUE) ;
    return 0 ;
```

此处的关键是:DialogBoxParam 的最后一个参数是作为 WM_INITDIALOG 消息中的 lParam 传递给对话框过程的。

对话框过程有两个基于 ABOUTBOX_DATA 结构的静态变量(一个结构和一个指向结构的指针):

```
static ABOUTBOX_DATA ad, * pad ;
```

在 AboutDlgProc 中,此定义代替了 iColor 和 iFigure 的定义。在 WM_INITDIALOG 消息的开始部分,对话框过程根据 lParam 设置了这两个变量的值:

```
pad = (ABOUTBOX_DATA *) lParam ;
ad = * pad ;
```

第一条语句中,pad 设置为 lParam 的指针。即 pad 实际是指向在 WndProc 定义的 ABOUTBOX_DATA 结构。第二个参数完成了从 WndProc 中的结构到 DlgProc 中的本地结构的字段到字段的复制。

现在,除了用户按下 OK 按钮时所用的代码以外,所有的 AboutDlgProc 都用 ad.iColor 和 ad.iFigure 替换 iFigure 和 iColor。这时,将本地结构的内容复制回 WndProc 中的结构:

```
case IDOK:
    * pad = ad ;
    EndDialog (hDlg, TRUE) ;
    return TRUE ;
```

11.1.10 Tab 停留位和组

在第九章,我们利用窗口子类化为 COLORS1 添加功能,使我们能够通过按下 Tab 键来

从一个滚动条转移到另一个滚动条。在对话框中，窗口子类化是不必要的，因为 Windows 完成了将输入焦点从一个控制移动到另一个控制的所有工作。尽管如此，您必须在对话框模板中使用 WS_TABSTOP 和 WS_GROUP 窗口风格达到此目的。对于所有要使用 Tab 键访问的控制，都要在其窗口风格中指定 WS_TABSTOP。

如果参见表 11-1，您就会注意到许多控制将 WS_TABSTOP 定义为默认风格，其他一些则没有将它作为默认风格。一般而言，不包含 WS_TABSTOP 风格的控制（特别是静态控制）不应该获取输入焦点，因为即使有了输入焦点，它们也不能完成操作。除非在处理 WM_INITDIALOG 消息时将输入焦点设置给一个特定的控制，并从消息中返回 FALSE；否则，Windows 将输入焦点设置为对话框内第一个具有 WS_TABSTOP 风格的控制。

Windows 给对话框添加的第二个键盘接口包括光标移动键，这种接口对于单选按钮有特殊的重要性。如果使用 Tab 键移动到某一组内当前选中的单选按钮，那么就需要使用光标移动键将输入焦点从该单选按钮移动到组内其他单选按钮上。使用 WS_GROUP 窗口风格即可获得这个功能。对于对话框模板中的特定控制序列，Windows 将使用光标移动键把输入焦点从第一个具有 WS_GROUP 风格的控制切换到下一个具有 WS_GROUP 风格的控制中。如果有必要，Windows 将从对话框的最后一个控制循环到第一个控制，以便找到组的结尾。

在默认设置下，控制 LTEXT、CTEXT、RTEXT 和 ICON 包含有 WS_GROUP 风格，这种风格方便地标记了组的结尾。您必须经常将 WS_GROUP 风格添加到其他类型的控制中。

让我们来看一看 ABOUT2.RC 中的对话框模板。4 个具有 WS_TABSTOP 风格的控制是每个组的第一个单选按钮（显式地包含）和两个下压按钮（默认设置）。在第一次激活对话框时，您可以使用 Tab 键在这 4 个控制之间移动。

在每组单选按钮中，您可以使用光标移动键切换输入焦点并更改复选标记。例如，Color 组合框的第一个单选按钮（Black）和 Figure 组合框都具有 WS_GROUP 风格。这意味着您可以用光标移动键将焦点从 Black 单选按钮移动到 Figure 分组框中。类似地，Figure 分组框的第一个单选按钮（Rectangle）和 DEFPUSHBUTTON 都具有 WS_GROUP 风格，所以您可以使用光标移动键在组内两个单选按钮 Rectangle 和 Ellipse 之间移动。两个下压按钮都有 WS_GROUP 风格，以阻止光标移动键在下压按钮具有输入焦点时起作用。

使用 ABOUT2 时，Windows 的对话框管理器在两组单选按钮中完成一些相当复杂的处理。正如所期望的那样，处于单选按钮组内时，光标移动键切换输入焦点，并给对话框过程发送 WM_COMMAND 消息。但是，当您改变了组内选中的单选按钮时，Windows

也给新选中的单选按钮赋予 WS_TABSTOP 风格。当您下一次使用 Tab 键切换到这一组后，Windows 将会把输入焦点设置为选中的单选按钮。

文本字段中的“&”将导致紧跟其后的字母以下划线显示，这就增加了另一种键盘接口，您可以通过按下划线字母来将输入焦点移动到任意单选按钮上。通过按 C 键（代表 Color 组合框）或者 F 键（代表 Figure 组合框），您可以将输入焦点移动到相应组内当前选中的单选按钮上。

尽管程序员通常让对话框管理器来完成这些工作，但是 Windows 提供了两个函数，以便程序员查找下一个或者前一个 Tab 键停留项或者组项。这些函数为

```
hwndCtrl = GetNextDlgTabItem (hDlg, hwndCtrl, bPrevious) ;
```

和

```
hwndCtrl = GetNextDlgGroupItem (hDlg, hwndCtrl, bPrevious) ;
```

如果 bPrevious 为 TRUE，那么函数返回前一个 Tab 键停留项或组项；如果为 FALSE，则返回下一个 Tab 键停留项或者组项。

11.1.11 在对话框上绘图

ABOUT2 还完成了一些相对而言不同寻常的事情，即在对话框上绘图。让我们来看一看它是怎样做的。在 ABOUT2.RC 的对话框模板内，使用位置和大小为我们想要绘图的区域定义了一块空白文本控制：

```
LTEXT "" IDC_PAINT, 114, 67, 72, 72
```

这个区域为 18 个字符宽和 9 个字符高。由于这个控制没有文本，所以窗口过程为“静态”类所做的工作只是在必须重绘这个子窗口控制时清除其背景。

在当前颜色或者图形选择发生改变，或者对话框自身获得一个 WM_PAINT 消息时，对话框过程调用 PaintTheBlock，这个函数在 ABOUT2.C 中：

```
PaintTheBlock (hCtrlBlock, iColor, iFigure) ;
```

在 AboutDlgProc 中，窗口句柄 hCtrlBlock 已经在处理 WM_INITDIALOG 消息时被设置：

```
hCtrlBlock = GetDlgItem (hDlg, IDD_PAINT) ;
```

下面是 PaintTheBlock 函数：

```
void PaintTheBlock (HWND hCtrl, int iColor, int iFigure)
{
     InvalidateRect (hCtrl, NULL, TRUE) ;
     UpdateWindow (hCtrl) ;
     PaintWindow (hCtrl, iColor, iFigure) ;
```

}

这个函数使得子窗口控制无效，并为控制窗口过程产生一个 WM_PAINT 消息，然后调用 ABOUT2 中的另一个函数 PaintWindow。

PaintWindow 函数获取一个设备描述表句柄，并将其放到 hCtrl 中，画出所选图形，根据所选颜色用一个着色画刷填充图形。子窗口控制的大小从 GetClientRect 获得。尽管对话框模板以字符为单位定义了控制的大小，但 GetClientRect 获取以像素点为单位的尺寸。您也可以使用函数 MapDialogRect 将对话框中的字符坐标转换为客户区中的像素点坐标。

我们并非真地绘制了对话框的客户区，实际上绘制的是子窗口控制的客户区。每当对话框得到一个 WM_PAINT 消息时，就令子窗口控制失效，并刷新它，使它确信现在其客户区又有效了，然后再在其上画图。

11.1.12 将其他函数用于对话框

大多数可以用于子窗口的函数也可以用于对话框中的控制。例如，如果您预感到有些不妙的话，那么可以使用 MoveWindow 在对话框内移动控制，强迫用户用鼠标来跟踪它们。

有时，您需要根据其他控制的设置来动态地启用或禁用对话框中的具体控制，这需要调用：

```
EnableWindow (hwndCtrl, bEnable) ;
```

当 bEnable 为 TRUE(非 0)时，启用该控制；当 bEnable 为 FALSE(0)时，禁用该控制。当控制被禁用时，它不再接收键盘或鼠标的输入，因此不要禁用一个拥有输入焦点的控制。

11.1.13 定义自己的控制

尽管 Windows 承担了许多维护对话框和子窗口控制的工作，但是它同时也给您提供了各种加入代码的方法。前面我们已经看到了在对话框上绘图的方法。您也可以使用第九章中讨论的窗口子类化来更改子窗口控制的操作。

您还可以定义自己的子窗口控制，并将它们用于对话框中。例如，假定您特别不喜欢普通的矩形下压按钮，而倾向于创建椭圆形下压按钮，那么您可以通过注册一个窗口类，并使用自己编制的窗口过程处理来自您所创建窗口的消息，从而创建椭圆形下压按钮。在 Developer Studio 中，您可以在与自定义控制相联系的 Properties 对话框中指定这个窗口类，这将转换成对话框模板中的 CONTROL 语句。程序 11-3 所示的 ABOUT3 程序正是这样做的。

程序 11-3 ABOUT3 程序

ABOUT3.C

```
/*-----------------------------------------
   ABOUT3.C — About Box Demo Program No. 3
              (c) Charles Petzold, 1998
  -----------------------------------------*/

#include <windows.h>
#include "resource.h"
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
BOOL    CALLBACK AboutDlgProc (HWND, UINT, WPARAM, LPARAM) ;
LRESULT CALLBACK EllipPushWndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("About3") ;
     MSG          msg ;
     HWND         hwnd ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (hInstance, szAppName) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = szAppName ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = EllipPushWndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = NULL ;
```

```
     wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground  = (HBRUSH) (COLOR_BTNFACE + 1) ;
     wndclass.lpszMenuName   = NULL ;
     wndclass.lpszClassName  = TEXT ("EllipPush") ;

     RegisterClass (&wndclass) ;

     hwnd = CreateWindow (szAppName, TEXT ("About Box Demo Program"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static HINSTANCE hInstance ;

     switch (message)
     {
     case WM_CREATE :
          hInstance = ((LPCREATESTRUCT) lParam)->hInstance ;
          return 0 ;

     case WM_COMMAND :
          switch (LOWORD (wParam))
          {
          case IDM_APP_ABOUT :
               DialogBox (hInstance, TEXT ("AboutBox"), hwnd, AboutDlgProc) ;
               return 0 ;
          }
          break ;

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
     }
```

```
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}

BOOL CALLBACK AboutDlgProc (HWND hDlg, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
     switch (message)
     {
     case WM_INITDIALOG :
          return TRUE ;

     case WM_COMMAND :
          switch (LOWORD (wParam))
          {
          case IDOK :
               EndDialog (hDlg, 0) ;
               return TRUE ;
          }
          break ;
     }
     return FALSE ;
}

LRESULT CALLBACK EllipPushWndProc (HWND hwnd, UINT message,
                                   WPARAM wParam, LPARAM lParam)
{
     TCHAR          szText[40] ;
     HBRUSH         hBrush ;
     HDC            hdc ;
     PAINTSTRUCT    ps ;
     RECT           rect ;

     switch (message)
     {
     case WM_PAINT :
          GetClientRect (hwnd, &rect) ;
          GetWindowText (hwnd, szText, sizeof (szText)) ;

          hdc = BeginPaint (hwnd, &ps) ;

          hBrush = CreateSolidBrush (GetSysColor (COLOR_WINDOW)) ;
          hBrush = (HBRUSH) SelectObject (hdc, hBrush) ;
          SetBkColor (hdc, GetSysColor (COLOR_WINDOW)) ;
          SetTextColor (hdc, GetSysColor (COLOR_WINDOWTEXT)) ;

          Ellipse (hdc, rect.left, rect.top, rect.right, rect.bottom) ;
          DrawText (hdc, szText, -1, &rect,
                    DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
```

```
        DeleteObject (SelectObject (hdc, hBrush)) ;

        EndPaint (hwnd, &ps) ;
        return 0 ;

    case WM_KEYUP :
        if (wParam != VK_SPACE)
            break ;
                                        // fall through
    case WM_LBUTTONUP :
        SendMessage (GetParent (hwnd), WM_COMMAND,
                GetWindowLong (hwnd, GWL_ID), (LPARAM) hwnd) ;
        return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

ABOUT3.RC (摘录)

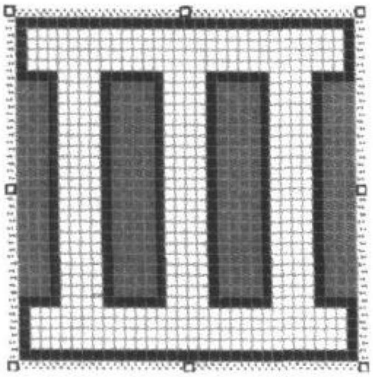
```
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
// Dialog

ABOUTBOX DIALOG DISCARDABLE 32, 32, 180, 100
STYLE DS_MODALFRAME | WS_POPUP
FONT 8, "MS Sans Serif"
BEGIN
    CONTROL         "OK",IDOK,"EllipPush",WS_GROUP | WS_TABSTOP,73,79,32,14
    ICON            "ABOUT3",IDC_STATIC,7,7,20,20
    CTEXT           "About3",IDC_STATIC,40,12,100,8
    CTEXT           "About Box Demo Program",IDC_STATIC,7,40,166,8
    CTEXT           "(c) Charles Petzold, 1998",IDC_STATIC,7,52,166,8
END

/////////////////////////////////////////////////////////////////////////////
// Menu

ABOUT3 MENU DISCARDABLE
BEGIN
    POPUP "&Help"
    BEGIN
        MENUITEM "&About About3...",            IDM_APP_ABOUT
    END
```

```
END

/////////////////////////////////////////////////////////////////////////////
// Icon

ABOUT3                  ICON       DISCARDABLE       "icon1.ico"
```

RESOURCE.H (摘录)

```
// Microsoft Developer Studio generated include file.
// Used by About3.rc

#define IDM_APP_ABOUT                  40001
#define IDC_STATIC                     -1
```

ABOUT3.ICO



我们所注册的窗口类叫做 EllipPush(“椭圆形下压按钮”)。在 Developer Studio 的对话框编辑器中,删除 Cancel 和 OK 按钮。要添加基于此窗口类的控件,请从 Controls 工具栏选择 Custom Control。在此控件的 Properties 对话框的 Class 域输入 EllipPush。在对话框模板中我们没有使用 DEFPUSHBUTTON 语句,而是用 CONTROL 语句来指定此窗口类:

```
CONTROL "OK" IDOK, "EllipPush", TABGRP, 64, 60, 32, 14
```

当在对话框中创建子窗口控件时,对话框管理器把这个窗口类用于 CreateWindow.调用中。

ABOUT3.C 程序在 WinMain 中注册了 EllipPush 窗口类:

```
wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
wndclass.lpfnWndProc   = EllipPushWndProc ;
wndclass.cbClsExtra    = 0 ;
wndclass.cbWndExtra    = 0 ;
wndclass.hInstance     = hInstance ;
```

```
wndclass.hIcon          = NULL ;
wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
wndclass.hbrBackground  = (HBRUSH) (COLOR_WINDOW + 1) ;
wndclass.lpszMenuName   = NULL ;
wndclass.lpszClassName  = TEXT ("EllipPush") ;

RegisterClass (&wndclass) ;
```

该窗口类指定窗口过程为 EllipPushWndProc,在 ABOUT3.C 中也是这样。

EllipPushWndProc 窗口过程只处理三种消息:WM_PAINT、WM_KEYUP 和 WM_LBUTTONUP。在处理 WM_PAINT 消息时,它从 GetClientRect 中获取窗口的大小,从 GetWindowText 中获取显示在下压按钮上的文本,用 Windows 函数 Ellipse 和 DrawText 来输出椭圆和文本。

WM_KEYUP 和 WM_LBUTTONUP 消息的处理非常简单:

```
case WM_KEYUP:
    if (wParam != VK_SPACE)
        break ;
                                    // fall through
case WM_LBUTTONUP :
    SendMessage (GetParent (hwnd), WM_COMMAND,
        GetWindowLong (hwnd, GWL_ID), (LPARAM) hwnd) ;
    return 0 ;
```

窗口过程使用 GetParent 来获取其父窗口(即对话框)的句柄,并发送一个 WM_COMMAND 消息,消息的 wParam 等于控制的 ID,这个 ID 是用 GetWindowLong 获取的。然后,对话框窗口过程将这个消息传给 ABOUT3 内的对话框过程,结果得到一个用户定制的下压按钮,如下页图 11-3 所示。您可以用同样的方法来创建其他定制对话框控制。

这就是全部要做的吗?其实不然。对于通常维护子窗口控制所需要的处理而言,EllipPushWndProc 只是一个空架子。例如,按钮不会像普通的下压按钮那样闪烁。要翻转下压按钮内的颜色,窗口过程必须处理 WM_KEYDOWN(来自 Spacebar 键)和 WM_LBUTTONDOWN 消息。窗口过程还必须在收到 WM_LBUTTONDOWN 消息时捕获鼠标,并且,如果当按钮还处于按下状态时,鼠标移到了子窗口的客户区之外,那么要释放鼠标捕获(并将按钮的内部颜色返回为正常状态)。只有在鼠标被捕获时释放该按钮,子窗口才会给其父窗口发送回一个 WM_COMMAND 消息。

EllipPushWndProc 也不处理 WM_ENABLE 消息。如上所述,对话框过程可以使用 EnableWindow 函数来禁用某窗口,于是,子窗口将显示灰色文本,而不再是黑色文本,以表示它已经被禁用,而不能再接收任何消息了。

如果子窗口控件的窗口过程需要为所创建的每个窗口存放各自不同的数据，那么它可以通过使用窗口类结构中的 cbWndExtra 值来做到。这样就在内部窗口结构中保留了空间，并可以用 SetWindowLong 和 GetWindowLong 来访问该数据。

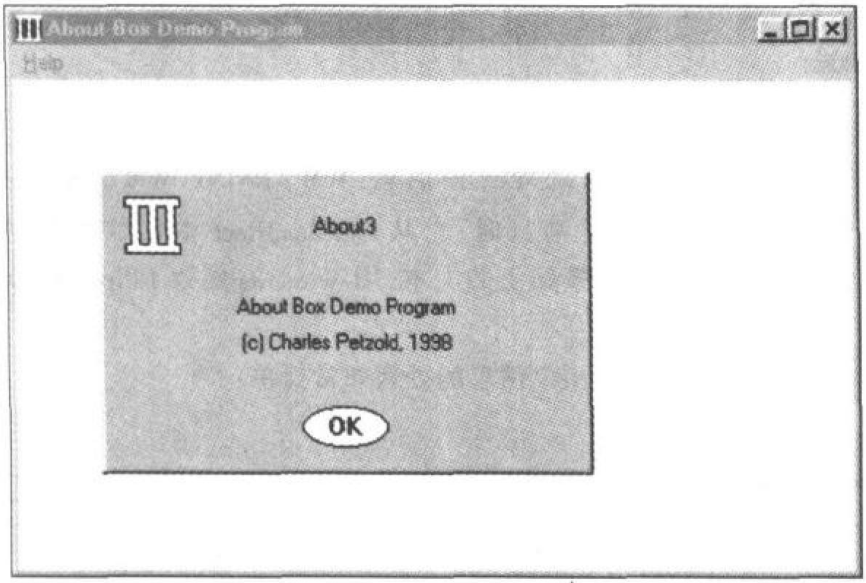


图 11-3　ABOUT3 创建的自定义下压按钮

11.2 非模态对话框

在本章的开始，我曾经说过对话框分为"模态的"和"非模态的"两种。现在我们已经研究过了这两种对话框中最常见的一种——模态对话框。模态对话框(不包括系统模态对话框)允许用户在对话框与其他程序之间进行切换。但是在清除模态对话框之前，用户不能切换到同一程序的另一个窗口。非模态对话框允许用户在对话框与其他程序之间进行切换，又可以在对话框与创建对话框的窗口之间进行切换。因此，非模态对话框与用户程序常见的普通弹出式窗口可能更相似。

当用户觉得让对话框保留片刻更加方便时，使用非模态对话框是合适的。例如，字处理器经常使用非模态对话框来做 Find 和 Change 对话框。如果 Find 对话框是模态的，那么用户必须从菜单中选择 Find，输入要查找的串，结束对话框，返回到文件中，然后再重复整个过程来查找同一串的其他匹配情况。允许用户在文件与对话框之间进行切换则会方便得多。

您已经看到，模态对话框是用 DialogBox 来创建的。只有在清除对话框之后，函数才返回值。在对话框过程内使用 EndDialog 调用来终止对话框，DialogBox 返回的是该调用的第二个参数的值。非模态对话框是使用 CreateDialog 来创建的，该函数使用的参数与 DialogBox 相同：

```
hDlgModeless = CreateDialog (hInstance, szTemplate,
                             hwndParent, DialogProc) ;
```

区别是 CreateDialog 函数立即返回对话框的窗口句柄，通常将这个窗口句柄存放到全局变量中。

尽管将 DialogBox 这一名字用于模态对话框，而 CreateDialog 用于非模态对话框是随意的，但是您可以通过非模态对话框与普通窗口类似这一点来记住这两个函数的区别。CreateDialog 可以令人想起 CreateWindow 函数来，后者创建的是普通窗口。

11.2.1 模态对话框与非模态对话框的区别

使用非模态对话框与使用模态对话框相似，但是也有一些重要的区别：

首先，非模态对话框通常包含一个标题栏和一个系统菜单框。当您在 Developer Studio 中创建对话框时，这些是默认选项。用于非模态对话框的对话框模板中的 STYLE 语句形如：

```
STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU | WS_VISIBLE
```

标题栏和系统菜单允许用户使用鼠标或者键盘将非模态对话框移动到另一个显示区域。对于模态对话框，通常无需提供标题栏和系统菜单，因为用户不能在其下面的窗口中做任何其他事情。

第二条重要的区别是：注意，在我们的示例 STYLE 语句中包含有 WS_VISIBLE 风格。在 Developer Studio 中，从 Dialog Properties 对话框的 More Styles 选项卡中选择此选项。如果省略了 WS_VISIBLE，那么您必须在 CreateDialog 调用之后调用 ShowWindow：

```
hDlgModeless = CreateDialog ( . . . ) ;
ShowWindow (hDlgModeless, SW_SHOW) ;
```

如果您既没有包含 WS_VISIBLE 风格，又没有调用 ShowWindow，那么非模态对话框不会被显示。如果忽略这个事实，那么习惯于模态对话框的程序员在第一次试图创建非模态对话框时，将会经常出现问题。

第三条区别：与模态对话框和消息框的消息不同，非模态对话框的消息要经过程序的消息队列。要将这些消息传送给对话框窗口过程，则必须更改消息队列。方法如下：当您使用 CreateDialog 创建非模态对话框时，应该将从调用中返回的对话框句柄保存在一个全局变量(如 hDlgModeless)中，并将消息循环更改为

```
while (GetMessage (&msg, NULL, 0, 0))
{
     if (hDlgModeless == 0 || !IsDialogMessage (hDlgModeless, &msg))
     {
          TranslateMessage (&msg) ;
```

```
        DispatchMessage (&msg) ;
    }
}
```

如果消息是发送给非模态对话框的，那么 IsDialogMessage 将它发送给对话框中窗口过程，并返回 TRUE(非 0)；否则，它将返回 FALSE(0)。只有 hDlgModeless 为 0 或者消息不是该对话框的消息时，才必须调用 TranslateMessage 和 DispatchMessage 函数。如果将键盘加速键用于您的程序窗口，那么消息循环将如下所示：

```
while (GetMessage (&msg, NULL, 0, 0))
{
    if (hDlgModeless == 0 || !IsDialogMessage (hDlgModeless, &msg))
    {
        if (!TranslateAccelerator (hwnd, hAccel, &msg))
        {
            TranslateMessage (&msg) ;
            DispatchMessage (&msg) ;
        }
    }
}
```

由于全局变量被初始化为 0，所以 hDlgModeless 将为 0，直到创建对话框为止，从而保证了不会使用无效的窗口句柄来调用 IsDialogMessage。在清除非模态对话框时，您也必须注意这一点，正如最后一点所说明的。

hDlgModeless 变量也可以由程序的其他部分使用，以便对非模态对话框是否存在加以验证。例如，程序中的其他窗口可以在 hDlgModeless 不等于 0 时给对话框发送消息。

最后一条重要的区别：使用 DestroyWindow 而不是 EndDialog 来结束非模态对话框。当您调用 DestroyWindow 后，将 hDlgModeless 全局变量设置为 0。

用户习惯于从系统菜单中选择 Close 来结束非模态对话框。尽管启用了 Close 选项，Windows 内的对话框窗口过程并不处理 WM_CLOSE 消息。您必须自己在对话框过程中处理它：

```
case WM_CLOSE :
    DestroyWindow (hDlg) ;
    hDlgModeless = NULL ;
    break ;
```

注意这两个窗口句柄之间的区别：DestroyWindow 的 hDlg 参数是传递给对话框过程的参数；hDlgModeless 是从 CreateDialog 返回的全局变量，程序在消息循环内检验它。

您也可以允许用户使用下压按钮来关闭非模态对话框，使用的逻辑与处理 WM_CLOSE 消息一样。对话框必须返回给创建它的窗口的任何数据都可以被存储在全局变量中。如果不喜欢使用全局变量，那么您也可以用 CreateDialogParam 来创建非模态对话

框，并按前面介绍的方法让它保存一个结构指针。

11.2.2 新的 COLORS 程序

第九章中所描述的 COLORS1 程序创建了 9 个子窗口，以便显示 3 个滚动条和 6 个文本项。那时候，这个程序还是所编制的程序中相当复杂的一个。如果将 COLORS1 转换为使用非模态对话框，则会使程序——特别是 WndProc 函数——变得令人难以置信的简单，修正后的 COLORS2 程序如程序 11-4 所示。

程序 11-4 COLORS2 程序

COLORS2.C

```
/*-------------------------------------------------------
   COLORS2.C——Version using Modeless Dialog Box
              (c) Charles Petzold, 1998
  -------------------------------------------------------*/
#include <windows.h>

LRESULT CALLBACK WndProc     (HWND, UINT, WPARAM, LPARAM) ;
BOOL    CALLBACK ColorScrDlg (HWND, UINT, WPARAM, LPARAM) ;

HWND hDlgModeless ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("Colors2") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = CreateSolidBrush (0L) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
```

```
                     szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Color Scroll"),
                          WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     hDlgModeless = CreateDialog (hInstance, TEXT ("ColorScrDlg"),
                                  hwnd, ColorScrDlg) ;
     while (GetMessage (&msg, NULL, 0, 0))
     {
          if (hDlgModeless == 0 || !IsDialogMessage (hDlgModeless, &msg))
          {
               TranslateMessage (&msg) ;
               DispatchMessage  (&msg) ;
          }
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     switch (message)
     {
     case WM_DESTROY :
          DeleteObject ((HGDIOBJ) SetClassLong (hwnd, GCL_HBRBACKGROUND,
                              (LONG) GetStockObject (WHITE_BRUSH))) ;
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}

BOOL CALLBACK ColorScrDlg (HWND hDlg, UINT message,
                           WPARAM wParam, LPARAM lParam)
{
     static int iColor[3] ;
     HWND       hwndParent, hCtrl ;
     int        iCtrlID, iIndex ;

     switch (message)
     {
```

```
case WM_INITDIALOG :
     for (iCtrlID = 10 ; iCtrlID < 13 ; iCtrlID++ )
     {
          hCtrl = GetDlgItem (hDlg, iCtrlID) ;
          SetScrollRange (hCtrl, SB_CTL, 0, 255, FALSE) ;
          SetScrollPos   (hCtrl, SB_CTL, 0, FALSE) ;
     }
     return TRUE ;

case WM_VSCROLL :
     hCtrl     = (HWND) lParam ;
     iCtrlID = GetWindowLong (hCtrl, GWL_ID) ;
     iIndex    = iCtrlID - 10 ;
     hwndParent = GetParent (hDlg) ;

     switch (LOWORD (wParam))
     {
     case SB_PAGEDOWN :
          iColor[iIndex] += 15 ;          // fall through
     case SB_LINEDOWN :
          iColor[iIndex] = min (255, iColor[iIndex] + 1) ;
          break ;
     case SB_PAGEUP :
          iColor[iIndex] -= 15 ;          // fall through
     case SB_LINEUP :
          iColor[iIndex] = max (0, iColor[iIndex] - 1) ;
          break ;
     case SB_TOP :
          iColor[iIndex] = 0 ;
          break ;
     case SB_BOTTOM :
          iColor[iIndex] = 255 ;
          break ;
     case SB_THUMBPOSITION :
     case SB_THUMBTRACK :
          iColor[iIndex] = HIWORD (wParam) ;
          break ;
     default :
          return FALSE ;
     }
     SetScrollPos  (hCtrl, SB_CTL,       iColor[iIndex], TRUE) ;
     SetDlgItemInt (hDlg, iCtrlID + 3,   iColor[iIndex], FALSE) ;

     DeleteObject ((HGDIOBJ) SetClassLong (hwndParent, GCL_HBRBACKGROUND,
                    (LONG) CreateSolidBrush (
                         RGB (iColor[0], iColor[1], iColor[2])))) ;
```

```
        InvalidateRect (hwndParent, NULL, TRUE) ;
        return TRUE ;
    }
    return FALSE ;
}
```

COLORS2.RC (摘录)

```
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
// Dialog

COLORSCRDLG DIALOG DISCARDABLE  16, 16, 120, 141
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION
CAPTION "Color Scroll Scrollbars"
FONT 8, "MS Sans Serif"
BEGIN
    CTEXT           "&Red",IDC_STATIC,8,8,24,8,NOT WS_GROUP
    SCROLLBAR       10,8,20,24,100,SBS_VERT | WS_TABSTOP
    CTEXT           "0",13,8,124,24,8,NOT WS_GROUP
    CTEXT           "&Green",IDC_STATIC,48,8,24,8,NOT WS_GROUP
    SCROLLBAR       11,48,20,24,100,SBS_VERT | WS_TABSTOP
    CTEXT           "0",14,48,124,24,8,NOT WS_GROUP
    CTEXT           "&Blue",IDC_STATIC,89,8,24,8,NOT WS_GROUP
    SCROLLBAR       12,89,20,24,100,SBS_VERT | WS_TABSTOP
    CTEXT           "0",15,89,124,24,8,NOT WS_GROUP
END
```

RESOURCE.H (摘录)

```
// Microsoft Developer Studio generated include file.
// Used by Colors2.rc

#define IDC_STATIC                      -1
```

原来的 COLORS1 程序所显示的滚动条大小是基于窗口大小的，而新程序在非模态对话框内以固定的尺寸来显示它们，如下图 11-4 所示。

当您创建对话框模板时，直接将 3 个滚动条的 ID 号分别设为 10、11 和 12，将显示滚动条当前值的 3 个静态文本域的 ID 号分别设为 13、14 和 15。将每个滚动条都设置为 Tab Stop 风格，而从所有的 6 个静态文本域中删除 Group 风格。

在 COLORS2 中，非模态对话框是在 WinMain 函数里创建的，紧跟在用于程序的主窗口的 ShowWindow 调用之后。注意，主窗口的窗口风格包含 WS_CLIPCHILDREN，这允

许程序无需擦除对话框就能够重绘主窗口。

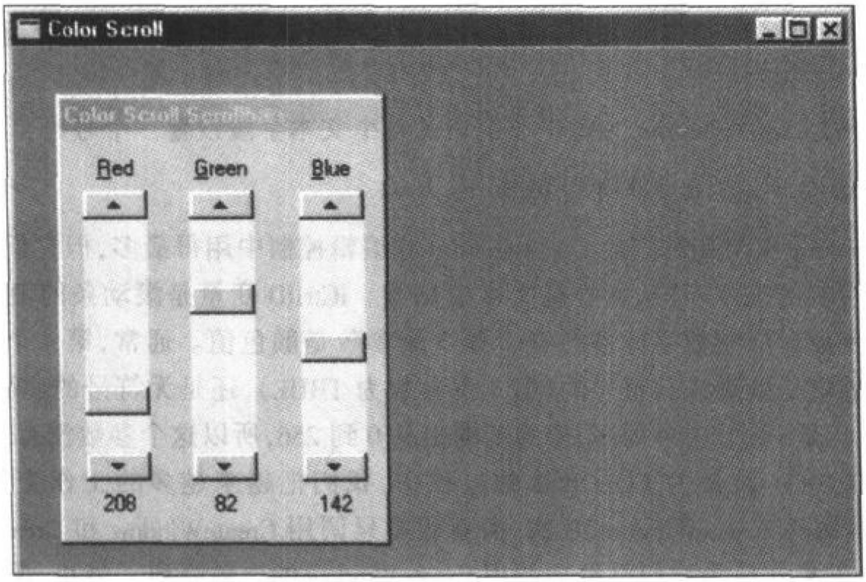


图 11-4 COLORS2 的屏幕显示

如上所述，从 CreateDialog 返回的对话框窗口句柄存放在全局变量 hDlgModeless 中，并在消息循环中被测试。不过，在这个程序中，不需要将句柄存放在全局变量中，也不需要在调用 IsDialogMessage 之前测试这个值。消息循环可以如下所示：

```
while (GetMessage (&msg, NULL, 0, 0))
{
     if (!IsDialogMessage (hDlgModeless, &msg))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
}
```

由于对话框是在程序进入消息循环前创建的，并且直到程序结束时才会被清除，所以 hDlgModeless 的值将总是有效的。我添加了如下的逻辑，以便您可能会往对话框的窗口过程中添加一段清除对话框的代码：

```
case WM_CLOSE :
     DestroyWindow (hDlg) ;
     hDlgModeless = NULL ;
     break ;
```

在原来的 COLORS1 程序中，SetWindowText 在使用 wsprintf 将三个数值标签转换为文本之后才设置它们的值。语句为：

```
wsprintf (szBuffer, TEXT ("%i "), color[i]) ;
SetWindowText (hwndValue[i], szBuffer) ;
```

i 的值为当前处理的滚动条的 ID 号，hwndValue 是一个数组，它包含颜色数值的三个静态文本子窗口的窗口句柄。

新版本使用 SetDlgItemInt 为每个子窗口的每个文本字段设置一个号码：

```
SetDlgItemInt (hDlg, iCtrlID + 3, color [iCtrlID], FALSE) ;
```

尽管 SetDlgItemInt 和与其对应的 GetDlgItemInt 在编辑控制中用得最多，但它们也可以用来设置其他控制的文本字段，如静态文本控制等。iCtrlID 变量是滚动条的 ID 号；给 ID 号加 3 使之变成对应的数字标签的 ID。第 3 个参数是颜色值。通常，第 4 个参数表明第 3 个参数的值是处理成有符号的(第 4 个参数为 TRUE)，还是无符号的(第 4 个参数为 FALSE)。但是，对于这个程序，值的范围是从 0 到 256，所以这个参数没有意义。

在将 COLORS1 转换为 COLORS2 的过程中，我们把越来越多的工作交给了 Windows。旧版本调用 CreateWindow 10 次；而新版本只调用 CreateWindow 和 CreateDialog 各一次。但是，如果您认为我们已经把调用 CreateWindow 的次数降到了最小，那么您就错了，请看下一个程序。

11.2.3 HEXCALC：窗口还是对话框？

HEXCALC 程序可能是偷懒的编程的典范之作，如程序 11-5 所示。这个程序根本不调用 CreateWindow，也不处理 WM_PAINT 消息，不获取设备描述表，也不处理鼠标消息。它只用了不到 150 行的源代码就构成了一个具有完整键盘和鼠标界面的 10 种功能的十六进制计算器。计算器如 501 页图 11-5 所示。

程序 11-5 HEXCALC 程序

HEXCALC.C

```
/*--------------------
    HEXCALC.C——Hexadecimal Calculator
               (c) Charles Petzold, 1998
  --------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR  szAppName[] = TEXT ("HexCalc") ;
```

```
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASS    wndclass ;

     wndclass.style          = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc    = WndProc ;
     wndclass.cbClsExtra     = 0 ;
     wndclass.cbWndExtra     = DLGWINDOWEXTRA ;    // Note!
     wndclass.hInstance      = hInstance ;
     wndclass.hIcon          = LoadIcon (hInstance, szAppName) ;
     wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground  = (HBRUSH) (COLOR_BTNFACE + 1) ;
     wndclass.lpszMenuName   = NULL ;
     wndclass.lpszClassName  = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateDialog (hInstance, szAppName, 0, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

void ShowNumber (HWND hwnd, UINT iNumber)
{
     TCHAR szBuffer[20] ;

     wsprintf (szBuffer, TEXT ("%X"), iNumber) ;
     SetDlgItemText (hwnd, VK_ESCAPE, szBuffer) ;
}

DWORD CalcIt (UINT iFirstNum, int iOperation, UINT iNum)
{
     switch (iOperation)
     {
     case '=' : return iNum ;
     case '+' : return iFirstNum + iNum ;
```

```
          case '-' : return iFirstNum - iNum ;
          case '*' : return iFirstNum * iNum ;
          case '&' : return iFirstNum & iNum ;
          case '|' : return iFirstNum | iNum ;
          case '^' : return iFirstNum ^ iNum ;
          case '<' : return iFirstNum << iNum ;
          case '>' : return iFirstNum >> iNum ;
          case '/' : return iNum ? iFirstNum / iNum: MAXDWORD ;
          case '%' : return iNum ? iFirstNum % iNum: MAXDWORD ;
          default  : return 0 ;
          }
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static BOOL  bNewNumber = TRUE ;
     static int   iOperation = '=' ;
     static UINT  iNumber, iFirstNum ;
     HWND         hButton ;

     switch (message)
     {
     case WM_KEYDOWN:                  // left arrow --> backspace
          if (wParam != VK_LEFT)
               break ;
          wParam = VK_BACK ;
                                       // fall through
     case WM_CHAR:
          if ((wParam = (WPARAM) CharUpper ((TCHAR *) wParam)) == VK_RETURN)
               wParam = '=' ;

          if (hButton = GetDlgItem (hwnd, wParam))
          {
               SendMessage (hButton, BM_SETSTATE, 1, 0) ;
               Sleep (100) ;
               SendMessage (hButton, BM_SETSTATE, 0, 0) ;
          }
          else
          {
               MessageBeep (0) ;
               break ;
          }
                                       // fall through
     case WM_COMMAND:
          SetFocus (hwnd) ;

          if (LOWORD (wParam) == VK_BACK)          // backspace
```

```
            ShowNumber (hwnd, iNumber /= 16) ;

        else if (LOWORD (wParam) == VK_ESCAPE)      // escape
            ShowNumber (hwnd, iNumber = 0) ;

        else if (isxdigit (LOWORD (wParam)))        // hex digit
        {
            if (bNewNumber)
            {
                iFirstNum = iNumber ;
                iNumber = 0 ;
            }
            bNewNumber = FALSE ;

            if (iNumber <= MAXDWORD >> 4)
                ShowNumber (hwnd, iNumber = 16 * iNumber + wParam -
                (isdigit (wParam) ? '0': 'A' - 10)) ;
            else
                MessageBeep (0) ;
        }
        else                                        // operation
        {
            if (!bNewNumber)
                ShowNumber (hwnd, iNumber =
                        CalcIt (iFirstNum, iOperation, iNumber)) ;
            bNewNumber = TRUE ;
            iOperation = LOWORD (wParam) ;
        }
        return 0 ;

    case WM_DESTROY:
        PostQuitMessage (0) ;
        return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

HEXCALC.RC (摘录)

```
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
// Icon

HEXCALC                 ICON        DISCARDABLE       "HexCalc.ico"
```

```
///////////////////////////////////////////////////////////////////////////////

# include "hexcalc.dlg"
```

HEXCALC.DLG

```
/*----------------
   HEXCALC.DLG dialog script
  ----------------*/

HexCalc DIALOG -1, -1, 102, 122
STYLE WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX
CLASS "HexCalc"
CAPTION "Hex Calculator"
{
  PUSHBUTTON "D",        68,   8,  24, 14, 14
  PUSHBUTTON "A",        65,   8,  40, 14, 14
  PUSHBUTTON "7",        55,   8,  56, 14, 14
  PUSHBUTTON "4",        52,   8,  72, 14, 14
  PUSHBUTTON "1",        49,   8,  88, 14, 14
  PUSHBUTTON "0",        48,   8, 104, 14, 14
  PUSHBUTTON "0",        27,  26,   4, 50, 14
  PUSHBUTTON "E",        69,  26,  24, 14, 14
  PUSHBUTTON "B",        66,  26,  40, 14, 14
  PUSHBUTTON "8",        56,  26,  56, 14, 14
  PUSHBUTTON "5",        53,  26,  72, 14, 14
  PUSHBUTTON "2",        50,  26,  88, 14, 14
  PUSHBUTTON "Back",      8,  26, 104, 32, 14
  PUSHBUTTON "C",        67,  44,  40, 14, 14
  PUSHBUTTON "F",        70,  44,  24, 14, 14
  PUSHBUTTON "9",        57,  44,  56, 14, 14
  PUSHBUTTON "6",        54,  44,  72, 14, 14
  PUSHBUTTON "3",        51,  44,  88, 14, 14
  PUSHBUTTON "+",        43,  62,  24, 14, 14
  PUSHBUTTON "-",        45,  62,  40, 14, 14
  PUSHBUTTON "*",        42,  62,  56, 14, 14
  PUSHBUTTON "/",        47,  62,  72, 14, 14
  PUSHBUTTON "%",        37,  62,  88, 14, 14
  PUSHBUTTON "Equals",   61,  62, 104, 32, 14
  PUSHBUTTON "&&",       38,  80,  24, 14, 14
  PUSHBUTTON "|",       124,  80,  40, 14, 14
  PUSHBUTTON "^",        94,  80,  56, 14, 14
  PUSHBUTTON "<",        60,  80,  72, 14, 14
  PUSHBUTTON ">",        62,  80,  88, 14, 14
}
```

HEXCALC.ICO

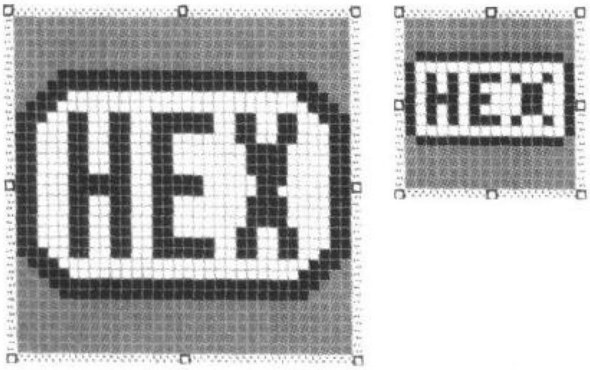


图 11-5 HEXCALC 的屏幕显示

HEXCALC 是一个通常的中缀表达式计算器，使用 C 语言句法进行计算。它对无符号 32 位整数进行加、减、乘、除和取余数运算，按位与、或及异或运算，还有左右位移运算。被 0 除将导致结果被设置为 FFFFFFFF。

在 HEXCALC 中既可以使用鼠标又可以使用键盘。从“单击”或者输入第一个数（最多 8 位十六进制数字）开始，然后输入运算符，然后是第二个数。接着，您可以通过单击 Equals 按钮，或者按下等号键或 Enter 键便可以显示运算结果。要更正输入的内容，您可以使用 Back 按钮、Backspace 键或者左箭头键。单击 display 框或者按下 Esc 键即可清除当前的输入。

HEXCALC 比较奇怪的一点是，屏幕上显示的窗口似乎是通常的重叠式窗口与非模态对话框的混合体。一方面，HEXCALC 的所有消息都在函数的 WndProc 中处理，这个函数与通常的窗口过程相似，该函数返回一个长整数，它处理 WM_DESTROY 消息，调

用 DefWindowProc，就像普通的窗口过程一样。另一方面，窗口是在具有 CreateDialog 调用的 WinMain 中创建的，CreateDialog 使用在 HEXCALC.DLG 中定义的对话框模板。那么，HEXCALC 到底是一个普通的重叠式窗口呢，还是一个非模态对话框呢？

简单的回答是，对话框就是窗口。通常，Windows 使用它自己内部的窗口过程处理对话框窗口的消息，然后，Windows 将这些消息传送给创建对话框的程序内的对话框过程。在 HEXCALC 中，我们让 Windows 使用对话框模板创建一个窗口，但是自己编程处理这个窗口的消息。

不幸的是，在 Developer Studio 的 Dialog Editor 中，对话框模板需要一些我们不能添加的东西。因此，对话框模板包含在 HEXCALC.DLG 文件中，而且需要手工输入。按下面的方法，您可以将一个文本文件添加到任意的项目中：从 File 菜单中选择 New，选择 Files 选项卡，然后从文件类型列表中选择 TextFile。像这样的文件——包含附加资源定义——需要包含在资源描述中。从 View 菜单中选择 Resource Includes。这显示一个对话框。在 Compile-time Directives 编辑域输入

```
#include "hexcalc.dlg"
```

这一行将插入到 HEXCALC.RC 资源描述中，像上面所显示的一样。

仔细看一下 HEXCALC.DLG 文件中的对话框模板，您将发现 HEXCALC 如何为对话框使用它自己的窗口过程。对话框模板的头部如下：

```
HexCalc DIALOG -1, -1, 102, 122
STYLE WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX
CLASS "HexCalc"
CAPTION "Hex Calculator"
```

注意，诸如 WS_OVERLAPPED 和 WS_MINIMIZEBOX 等标识符，我们可以将它们用在 CreateWindow 调用中，以创建普通的窗口。CLASS 语句是这个对话框与曾经创建过的对话框之间最重要的区别（而且它也是 Developer Studio 中的 Dialog Editor 不允许我们指定的）。当前面的对话框模板省略了这个语句时，Windows 为对话框注册一个窗口类，并使用它自己的窗口过程处理对话框消息。这里，包含 CLASS 语句就告诉 Windows 将消息发送到其他地方——具体地说，就是发送到在 HexCalc 窗口类中指定的窗口过程。

HexCalc 窗口类是在 HEXCALC 的 WinMain 函数中注册的，就像普通窗口的窗口类一样。但是，请注意这个十分重要的区别：WNDCLASS 结构的 cbWndExtra 字段设置为 DLGWINDOWEXTRA。对于您自己注册的对话框过程，这是必需的。

在注册窗口类之后，WinMain 调用 CreateDialog：

```
hwnd = CreateDialog (hInstance, szAppName, 0, NULL) ;
```

第二个参数（串“HexCalc”）是对话框模板的名称。第三个参数通常是父窗口的窗口句

柄，此处窗口没有父窗口，所以设置为0。最后一个参数，通常是对话框过程的地址，这里不需要，因为 Windows 不会处理这些消息，因而也不会将消息发送给对话框过程。

这个 CreateDialog 调用与对话框模板一起，被 Windows 有效地转换为一个 CreateWindow 调用。该 CreateWindow 调用的功能与下面的调用等价：

```
hwnd = CreateWindow (TEXT ("HexCalc"), TEXT ("Hex Calculator "),
          WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX,
          CW_USEDEFAULT, CW_USEDEFAULT,
          102 * 4 / cxChar, 122 * 8 / cyChar,
          NULL, NULL, hInstance, NULL) ;
```

其中，cxChar 和 cyChar 变量分别是系统字体字符的宽度和高度。

通过让 Windows 来进行 CreateWindow 调用而收获甚丰：Windows 不会在创建一个弹出式窗口后就停止，它还会为对话框模板中定义的其他29个子窗口下压按钮控制调用 CreateWindow。所有这些控制都给父窗口的窗口过程发送 WM_COMMAND 消息，该过程正是 WndProc。这是创建一个包含许多子窗口的窗口的很好技巧。

下面是使 HEXCALC 的代码大小下降到最少的另一种方法：或许您会注意到 HEXCALC 没有头文件，头文件中通常包含对话框模板中需要为所有子窗口控制定义的标识符。我们之所以可以不要这个文件，是因为每个下压按钮控制的 ID 号设置为出现在控制上的文本的 ASCII 码，这意味着，WndProc 可以完全相同地对待 WM_COMMAND 消息和 WM_CHAR 消息。在每种情况下，wParam 的低字都是按钮的 ASCII 码。

当然，对键盘消息进行一些处理是必要的。WndProc 捕获 WM_KEYDOWN 消息，将左箭头键转换为 Backspace 键。在处理 WM_CHAR 消息时，WndProc 将字符代码转换为大写，Enter 键转换为等号键的 ASCII 码。

WM_CHAR 消息的有效性是通过调用 GetDlgItem 来检验的。如果 GetDlgItem 函数返回0，那么键盘字符不是对话框模板中定义的 ID 号之一。如果字符是其中的一个 ID，则通过给相应的按钮发送一对 BM_SETSTATE 消息，来使之闪烁：

```
if (hButton = GetDlgItem (hwnd, wParam))
{
     SendMessage (hButton, BM_SETSTATE, 1, 0) ;
     Sleep (100) ;
     SendMessage (hButton, BM_SETSTATE, 0, 0) ;
}
```

这样做，用最小的代价，给 HEXCALC 的键盘界面增色不少。Sleep 函数将程序挂起100毫秒。这会防止按钮被“单击”得太快，以至不能被人注意。

当 WndProc 处理 WM_COMMAND 消息时，它总是将输入焦点设置给父窗口：

```
case WM_COMMAND :
```

```
SetFocus (hwnd) ;
```

否则,一旦使用鼠标单击某按钮,输入焦点就会切换到该按钮上。

11.3 通用对话框

Windows 的一个主要目的是推进标准的用户界面。对许多常用的菜单项来说,这发生得很快,几乎所有软件生产商都采用 Alt-File-Open 选择来打开一个文件。然而,实际的打开文件的对话框却常常有很大的不同。

从 Windows 3.1 开始,对这个问题有了一个可用的解决方案,这是一种被称为“通用对话框库”的增强方案。这个库由几个函数组成,这些函数激活标准对话框来进行打开和保存文件、查找和替换、选择颜色、选择字体(我将在本章讨论所有这些内容),以及打印(我将在第十三章讨论)。

为了使用这些函数,基本上都要初始化某一结构的各个字段,并将该结构的指针传送给通用对话框库的某个函数,该函数创建并显示对话框。当用户关闭对话框时,被调用的函数将控制返回给程序,您可以从传送给它的结构中获取信息。

在使用通用对话框库的任何 C 源代码文件中,您都需要包含 COMMDLG.H 头文件。通用对话框的文档在/Platform SDK/User Interface Services/User Input/Common Dialog Box Library 中。

11.3.1 增强 POPPAD

当我们向第十章的 POPPAD 中添加菜单时,还有几个标准菜单项没有实现。现在我们已经准备好在 POPPAD 中添加打开文件、读入文件,以及在磁盘上保存编辑过的文件的功能。在处理中,我们还将在 POPPAD 中添加选择字体,以及查找和替换功能。

实现 POPPAD3 程序的文件如程序 11-6 所示。

程序 11-6 POPPAD3 程序

POPPAD.C

```
/*---------------------------------------
   POPPAD.C——Popup Editor
             (c) Charles Petzold, 1998
  ---------------------------------------*/

#include <windows.h>
#include <commdlg.h>
#include "resource.h"

#define EDITID   1
```

```
#define UNTITLED TEXT ("(untitled)")

LRESULT CALLBACK WndProc      (HWND, UINT, WPARAM, LPARAM) ;
BOOL    CALLBACK AboutDlgProc (HWND, UINT, WPARAM, LPARAM) ;

     // Functions in POPFILE.C

void PopFileInitialize (HWND) ;
BOOL PopFileOpenDlg(HWND, PTSTR, PTSTR) ;
BOOL PopFileSaveDlg(HWND, PTSTR, PTSTR) ;
BOOL PopFileRead   (HWND, PTSTR) ;
BOOL PopFileWrite  (HWND, PTSTR) ;

     // Functions in POPFIND.C

HWND PopFindFindDlg      (HWND) ;
HWND PopFindReplaceDlg   (HWND) ;
BOOL PopFindFindText     (HWND, int *, LPFINDREPLACE) ;
BOOL PopFindReplaceText  (HWND, int *, LPFINDREPLACE) ;
BOOL PopFindNextText     (HWND, int * ) ;
BOOL PopFindValidFind    (void) ;

     // Functions in POPFONT.C

void PopFontInitialize       (HWND) ;
BOOL PopFontChooseFont       (HWND) ;
void PopFontSetFont          (HWND) ;
void PopFontDeinitialize     (void) ;
     // Functions in POPPRNT.C

BOOL PopPrntPrintFile (HINSTANCE, HWND, HWND, PTSTR) ;

     // Global variables

static HWND  hDlgModeless ;
static TCHAR szAppName[] = TEXT ("PopPad") ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     MSG       msg ;
     HWND      hwnd ;
     HACCEL    hAccel ;
     WNDCLASS  wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
```

```
     wndclass.cbWndExtra      = 0 ;
     wndclass.hInstance       = hInstance ;
     wndclass.hIcon           = LoadIcon (hInstance, szAppName) ;
     wndclass.hCursor         = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground   = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName    = szAppName ;
     wndclass.lpszClassName   = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, NULL,
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, szCmdLine) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;
     hAccel = LoadAccelerators (hInstance, szAppName) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          if (hDlgModeless == NULL || !IsDialogMessage (hDlgModeless, &msg))
          {
               if (!TranslateAccelerator (hwnd, hAccel, &msg))
               {
                    TranslateMessage (&msg) ;
                    DispatchMessage (&msg) ;
               }
          }
     }
     return msg.wParam ;
}

void DoCaption (HWND hwnd, TCHAR * szTitleName)
{
     TCHAR szCaption[64 + MAX_PATH] ;

     wsprintf (szCaption, TEXT ("%s - %s"), szAppName,
               szTitleName[0] ? szTitleName : UNTITLED) ;

     SetWindowText (hwnd, szCaption) ;
}
```

```
void OkMessage (HWND hwnd, TCHAR * szMessage, TCHAR * szTitleName)
{
     TCHAR szBuffer[64 + MAX_PATH] ;

     wsprintf (szBuffer, szMessage, szTitleName[0] ? szTitleName : UNTITLED) ;

     MessageBox (hwnd, szBuffer, szAppName, MB_OK | MB_ICONEXCLAMATION) ;
}

short AskAboutSave (HWND hwnd, TCHAR * szTitleName)
{
     TCHAR szBuffer[64 + MAX_PATH] ;
     int   iReturn ;

     wsprintf (szBuffer, TEXT ("Save current changes in %s?"),
               szTitleName[0] ? szTitleName : UNTITLED) ;

     iReturn = MessageBox (hwnd, szBuffer, szAppName,
                           MB_YESNOCANCEL | MB_ICONQUESTION) ;
     if (iReturn == IDYES)
       if (!SendMessage (hwnd, WM_COMMAND, IDM_FILE_SAVE, 0))
            iReturn = IDCANCEL ;

     return iReturn ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static BOOL         bNeedSave = FALSE ;
     static HINSTANCE    hInst ;
     static HWND         hwndEdit ;
     static int          iOffset ;
     static TCHAR        szFileName[MAX_PATH], szTitleName[MAX_PATH] ;
     static UINT         messageFindReplace ;
     int                 iSelBeg, iSelEnd, iEnable ;
     LPFINDREPLACE       pfr ;

     switch (message)
     {
     case WM_CREATE:
          hInst = ((LPCREATESTRUCT) lParam) -> hInstance ;

               // Create the edit control child window

          hwndEdit = CreateWindow (TEXT ("edit"), NULL,
                              WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL |
```

```
                            WS_BORDER | ES_LEFT | ES_MULTILINE |
                            ES_NOHIDESEL | ES_AUTOHSCROLL | ES_AUTOVSCROLL,
                            0, 0, 0, 0,
                            hwnd, (HMENU) EDITID, hInst, NULL) ;

          SendMessage (hwndEdit, EM_LIMITTEXT, 32000, 0L) ;

               // Initialize common dialog box stuff

          PopFileInitialize (hwnd) ;
          PopFontInitialize (hwndEdit) ;

          messageFindReplace = RegisterWindowMessage (FINDMSGSTRING) ;

          DoCaption (hwnd, szTitleName) ;
          return 0 ;
     case WM_SETFOCUS:
          SetFocus (hwndEdit) ;
          return 0 ;

     case WM_SIZE:
          MoveWindow (hwndEdit, 0, 0, LOWORD (lParam), HIWORD (lParam), TRUE) ;
          return 0 ;

     case WM_INITMENUPOPUP:
          switch (lParam)
          {
          case 1:             // Edit menu

                    // Enable Undo if edit control can do it

               EnableMenuItem ((HMENU) wParam, IDM_EDIT_UNDO,
                    SendMessage (hwndEdit, EM_CANUNDO, 0, 0L) ?
                                        MF_ENABLED : MF_GRAYED) ;

                    // Enable Paste if text is in the clipboard

               EnableMenuItem ((HMENU) wParam, IDM_EDIT_PASTE,
                    IsClipboardFormatAvailable (CF_TEXT) ?
                                   MF_ENABLED : MF_GRAYED) ;

                    // Enable Cut, Copy, and Del if text is selected

               SendMessage (hwndEdit, EM_GETSEL, (WPARAM) &iSelBeg,
                                                 (LPARAM) &iSelEnd) ;

               iEnable = iSelBeg != iSelEnd ? MF_ENABLED : MF_GRAYED ;
```

```
                EnableMenuItem ((HMENU) wParam, IDM_EDIT_CUT,   iEnable) ;
                EnableMenuItem ((HMENU) wParam, IDM_EDIT_COPY,  iEnable) ;
                EnableMenuItem ((HMENU) wParam, IDM_EDIT_CLEAR, iEnable) ;
                break ;

           case 2:             // Search menu

                // Enable Find, Next, and Replace if modeless
                //   dialogs are not already active

                iEnable = hDlgModeless == NULL ?
                               MF_ENABLED : MF_GRAYED ;
                EnableMenuItem ((HMENU) wParam, IDM_SEARCH_FIND,    iEnable) ;
                EnableMenuItem ((HMENU) wParam, IDM_SEARCH_NEXT,    iEnable) ;
                EnableMenuItem ((HMENU) wParam, IDM_SEARCH_REPLACE, iEnable) ;
                break ;
           }
           return 0 ;

      case WM_COMMAND:
                // Messages from edit control

           if (lParam && LOWORD (wParam) == EDITID)
           {
                switch (HIWORD (wParam))
                {
                case EN_UPDATE :
                     bNeedSave = TRUE ;
                     return 0 ;

                case EN_ERRSPACE :
                case EN_MAXTEXT :
                     MessageBox (hwnd, TEXT ("Edit control out of space."),
                                 szAppName, MB_OK | MB_ICONSTOP) ;
                     return 0 ;
                }
                break ;
           }

           switch (LOWORD (wParam))
           {
                     // Messages from File menu

           case IDM_FILE_NEW:
                if (bNeedSave && IDCANCEL == AskAboutSave (hwnd, szTitleName))
                     return 0 ;
```

```
          SetWindowText (hwndEdit, TEXT ("\0")) ;
          szFileName[0] = '\0' ;
          szTitleName[0] = '\0' ;
          DoCaption (hwnd, szTitleName) ;
          bNeedSave = FALSE ;
          return 0 ;

     case IDM_FILE_OPEN:
          if (bNeedSave && IDCANCEL == AskAboutSave (hwnd, szTitleName))
               return 0 ;
          if (PopFileOpenDlg (hwnd, szFileName, szTitleName))
          {
               if (!PopFileRead (hwndEdit, szFileName))
               {
                    OkMessage (hwnd, TEXT ("Could not read file %s!"),
                                      szTitleName) ;
                    szFileName[0] = '\0' ;
                    szTitleName[0] = '\0' ;
               }
          }

          DoCaption (hwnd, szTitleName) ;
          bNeedSave = FALSE ;
          return 0 ;

     case IDM_FILE_SAVE:
          if (szFileName[0])
          {
               if (PopFileWrite (hwndEdit, szFileName))
               {
                    bNeedSave = FALSE ;
                    return 1 ;
               }
               else
               {
                    OkMessage (hwnd, TEXT ("Could not write file %s"),
                                      szTitleName) ;
                    return 0 ;
               }
          }
                                        // fall through
     case IDM_FILE_SAVE_AS:
          if (PopFileSaveDlg (hwnd, szFileName, szTitleName))
          {
               DoCaption (hwnd, szTitleName) ;

               if (PopFileWrite (hwndEdit, szFileName))
```

```
                {
                     bNeedSave = FALSE ;
                     return 1 ;
                }
                else
                {
                     OkMessage (hwnd, TEXT ("Could not write file %s"),
                                szTitleName) ;
                     return 0 ;
                }
           }
           return 0 ;

      case IDM_FILE_PRINT:
           if (!PopPrntPrintFile (hInst, hwnd, hwndEdit, szTitleName))
                OkMessage (hwnd, TEXT ("Could not print file %s"),
                                szTitleName) ;
           return 0 ;

      case IDM_APP_EXIT:
           SendMessage (hwnd, WM_CLOSE, 0, 0) ;
           return 0 ;

                // Messages from Edit menu

      case IDM_EDIT_UNDO:
           SendMessage (hwndEdit, WM_UNDO, 0, 0) ;
           return 0 ;

      case IDM_EDIT_CUT:
           SendMessage (hwndEdit, WM_CUT, 0, 0) ;
           return 0 ;

      case IDM_EDIT_COPY:
           SendMessage (hwndEdit, WM_COPY, 0, 0) ;
           return 0 ;

      case IDM_EDIT_PASTE:
           SendMessage (hwndEdit, WM_PASTE, 0, 0) ;
           return 0 ;

      case IDM_EDIT_CLEAR:
           SendMessage (hwndEdit, WM_CLEAR, 0, 0) ;
           return 0 ;

      case IDM_EDIT_SELECT_ALL:
           SendMessage (hwndEdit, EM_SETSEL, 0, -1) ;
```

```
          return 0 ;

               // Messages from Search menu

     case IDM_SEARCH_FIND:
          SendMessage (hwndEdit, EM_GETSEL, 0, (LPARAM) &iOffset) ;
          hDlgModeless = PopFindFindDlg (hwnd) ;
          return 0 ;

     case IDM_SEARCH_NEXT:
          SendMessage (hwndEdit, EM_GETSEL, 0, (LPARAM) &iOffset) ;

          if (PopFindValidFind ())
               PopFindNextText (hwndEdit, &iOffset) ;
          else
               hDlgModeless = PopFindFindDlg (hwnd) ;

          return 0 ;

     case IDM_SEARCH_REPLACE:
          SendMessage (hwndEdit, EM_GETSEL, 0, (LPARAM) &iOffset) ;
          hDlgModeless = PopFindReplaceDlg (hwnd) ;
          return 0 ;

     case IDM_FORMAT_FONT:
          if (PopFontChooseFont (hwnd))
               PopFontSetFont (hwndEdit) ;

          return 0 ;

               // Messages from Help menu

     case IDM_HELP:
          OkMessage (hwnd, TEXT ("Help not yet implemented!"),
                           TEXT ("\0")) ;
          return 0 ;

     case IDM_APP_ABOUT:
          DialogBox (hInst, TEXT ("AboutBox"), hwnd, AboutDlgProc) ;
          return 0 ;
     }
     break ;

case WM_CLOSE:
     if (!bNeedSave || IDCANCEL != AskAboutSave (hwnd, szTitleName))
          DestroyWindow (hwnd) ;
```

```
            return 0 ;
        case WM_QUERYENDSESSION :
            if (!bNeedSave || IDCANCEL != AskAboutSave (hwnd, szTitleName))
                return 1 ;

            return 0 ;

        case WM_DESTROY:
            PopFontDeinitialize () ;
            PostQuitMessage (0) ;
            return 0 ;

        default:
                // Process "Find-Replace" messages

            if (message == messageFindReplace)
            {
                pfr = (LPFINDREPLACE) lParam ;

                if (pfr -> Flags & FR_DIALOGTERM)
                    hDlgModeless = NULL ;

                if (pfr -> Flags & FR_FINDNEXT)
                    if (!PopFindFindText (hwndEdit, &iOffset, pfr))
                        OkMessage (hwnd, TEXT ("Text not found!"),
                                         TEXT ("\0")) ;

                if (pfr -> Flags & FR_REPLACE || pfr -> Flags & FR_REPLACEALL)
                    if (!PopFindReplaceText (hwndEdit, &iOffset, pfr))
                        OkMessage (hwnd, TEXT ("Text not found!"),
                                         TEXT ("\0")) ;

                if (pfr -> Flags & FR_REPLACEALL)
                    while (PopFindReplaceText (hwndEdit, &iOffset, pfr)) ;

                return 0 ;
            }
            break ;
        }
        return DefWindowProc (hwnd, message, wParam, lParam) ;
}

BOOL CALLBACK AboutDlgProc (HWND hDlg, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
        switch (message)
        {
        case WM_INITDIALOG:
```

```
          return TRUE ;

     case WM_COMMAND:
          switch (LOWORD (wParam))
          {
          case IDOK:
               EndDialog (hDlg, 0) ;
               return TRUE ;
          }
          break ;
     }
     return FALSE ;
}
```

POPFILE.C

```
/*---------------------------------------------
     POPFILE.C -- Popup Editor File Functions
  ---------------------------------------------*/

#include <windows.h>
#include <commdlg.h>

static OPENFILENAME ofn ;

void PopFileInitialize (HWND hwnd)
{
     static TCHAR szFilter[] = TEXT ("Text Files (*.TXT)\0*.txt\0")  \
                               TEXT ("ASCII Files (*.ASC)\0*.asc\0") \
                               TEXT ("All Files (*.*)\0*.*\0\0") ;

     ofn.lStructSize       = sizeof (OPENFILENAME) ;
     ofn.hwndOwner         = hwnd ;
     ofn.hInstance         = NULL ;
     ofn.lpstrFilter       = szFilter ;
     ofn.lpstrCustomFilter = NULL ;
     ofn.nMaxCustFilter    = 0 ;
     ofn.nFilterIndex      = 0 ;
     ofn.lpstrFile         = NULL ;          // Set in Open and Close functions
     ofn.nMaxFile          = MAX_PATH ;
     ofn.lpstrFileTitle    = NULL ;          // Set in Open and Close functions
     ofn.nMaxFileTitle     = MAX_PATH ;
     ofn.lpstrInitialDir   = NULL ;
     ofn.lpstrTitle        = NULL ;
     ofn.Flags             = 0 ;             // Set in Open and Close functions
     ofn.nFileOffset       = 0 ;
     ofn.nFileExtension    = 0 ;
     ofn.lpstrDefExt       = TEXT ("txt") ;
     ofn.lCustData         = 0L ;
```

```
     ofn.lpfnHook           = NULL ;
     ofn.lpTemplateName     = NULL ;
}

BOOL PopFileOpenDlg (HWND hwnd, PTSTR pstrFileName, PTSTR pstrTitleName)
{
     ofn.hwndOwner          = hwnd ;
     ofn.lpstrFile          = pstrFileName ;
     ofn.lpstrFileTitle     = pstrTitleName ;
     ofn.Flags              = OFN_HIDEREADONLY | OFN_CREATEPROMPT ;

     return GetOpenFileName (&ofn) ;
}

BOOL PopFileSaveDlg (HWND hwnd, PTSTR pstrFileName, PTSTR pstrTitleName)
{
     ofn.hwndOwner          = hwnd ;
     ofn.lpstrFile          = pstrFileName ;
     ofn.lpstrFileTitle     = pstrTitleName ;
     ofn.Flags              = OFN_OVERWRITEPROMPT ;

     return GetSaveFileName (&ofn) ;
}

BOOL PopFileRead (HWND hwndEdit, PTSTR pstrFileName)
{
     BYTE      bySwap ;
     DWORD     dwBytesRead ;
     HANDLE    hFile ;
     int       i, iFileLength, iUniTest ;
     PBYTE     pBuffer, pText, pConv ;

          // Open the file.
     if (INVALID_HANDLE_VALUE ==
               (hFile = CreateFile (pstrFileName, GENERIC_READ, FILE_SHARE_READ,
                                    NULL, OPEN_EXISTING, 0, NULL)))
          return FALSE ;

          // Get file size in bytes and allocate memory for read.
          // Add an extra two bytes for zero termination.

     iFileLength = GetFileSize (hFile, NULL) ;
     pBuffer = malloc (iFileLength + 2) ;

          // Read file and put terminating zeros at end.

     ReadFile (hFile, pBuffer, iFileLength, &dwBytesRead, NULL) ;
     CloseHandle (hFile) ;
```

```
     pBuffer[iFileLength] = '\0' ;
     pBuffer[iFileLength + 1] = '\0' ;

          // Test to see if the text is Unicode

     iUniTest = IS_TEXT_UNICODE_SIGNATURE | IS_TEXT_UNICODE_REVERSE_SIGNATURE ;

     if (IsTextUnicode (pBuffer, iFileLength, &iUniTest))
     {
          pText = pBuffer + 2 ;
          iFileLength -= 2 ;

          if (iUniTest & IS_TEXT_UNICODE_REVERSE_SIGNATURE)
          {
               for (i = 0 ; i < iFileLength / 2 ; i++)
               {
                    bySwap = ((BYTE *) pText) [2 * i] ;
                    ((BYTE *) pText) [2 * i] = ((BYTE *) pText) [2 * i + 1] ;
                    ((BYTE *) pText) [2 * i + 1] = bySwap ;
               }
          }

               // Allocate memory for possibly converted string

          pConv = malloc (iFileLength + 2) ;

               // If the edit control is not Unicode, convert Unicode text to
               // non-Unicode (i.e., in general, wide character).

#ifndef UNICODE
          WideCharToMultiByte (CP_ACP, 0, (PWSTR) pText, -1, pConv,
                               iFileLength + 2, NULL, NULL) ;

               // If the edit control is Unicode, just copy the string
#else
          lstrcpy ((PTSTR) pConv, (PTSTR) pText) ;
#endif

     }
     else      // the file is not Unicode
     {
          pText = pBuffer ;

               // Allocate memory for possibly converted string.

          pConv = malloc (2 * iFileLength + 2) ;

               // If the edit control is Unicode, convert ASCII text.
```

```
# ifdef UNICODE
          MultiByteToWideChar (CP_ACP, 0, pText, -1, (PTSTR) pConv,
                               iFileLength + 1) ;

               // If not, just copy buffer
# else
          lstrcpy ((PTSTR) pConv, (PTSTR) pText) ;
# endif
     }

     SetWindowText (hwndEdit, (PTSTR) pConv) ;
     free (pBuffer) ;
     free (pConv) ;

     return TRUE ;
}

BOOL PopFileWrite (HWND hwndEdit, PTSTR pstrFileName)
{
     DWORD    dwBytesWritten ;
     HANDLE   hFile ;
     int      iLength ;
     PTSTR    pstrBuffer ;
     WORD     wByteOrderMark = 0xFEFF ;
          // Open the file, creating it if necessary

     if (INVALID_HANDLE_VALUE ==
               (hFile = CreateFile (pstrFileName, GENERIC_WRITE, 0,
                                    NULL, CREATE_ALWAYS, 0, NULL)))
          return FALSE ;

          // Get the number of characters in the edit control and allocate
          // memory for them.

     iLength = GetWindowTextLength (hwndEdit) ;
     pstrBuffer = (PTSTR) malloc ((iLength + 1) * sizeof (TCHAR)) ;

     if (!pstrBuffer)
     {
          CloseHandle (hFile) ;
          return FALSE ;
     }

               // If the edit control will return Unicode text, write the
               // byte order mark to the file.

# ifdef UNICODE
     WriteFile (hFile, &wByteOrderMark, 2, &dwBytesWritten, NULL) ;
```

```
#endif

          // Get the edit buffer and write that out to the file.

     GetWindowText (hwndEdit, pstrBuffer, iLength + 1) ;
     WriteFile (hFile, pstrBuffer, iLength * sizeof (TCHAR),
               &dwBytesWritten, NULL) ;

     if ((iLength * sizeof (TCHAR)) != (int) dwBytesWritten)
     {
          CloseHandle (hFile) ;
          free (pstrBuffer) ;
          return FALSE ;
     }

     CloseHandle (hFile) ;
     free (pstrBuffer) ;

     return TRUE ;
}
```

POPFIND.C

```
/*--------------------------------------------------------
     POPFIND.C——Popup Editor Search and Replace Functions
  --------------------------------------------------------*/

#include <windows.h>
#include <commdlg.h>
#include <tchar.h>            // for _tcsstr (strstr for Unicode & non-Unicode)

#define MAX_STRING_LEN   256

static TCHAR szFindText [MAX_STRING_LEN] ;
static TCHAR szReplText [MAX_STRING_LEN] ;

HWND PopFindFindDlg (HWND hwnd)
{
     static FINDREPLACE fr ;       // must be static for modeless dialog!!!

     fr.lStructSize      = sizeof (FINDREPLACE) ;
     fr.hwndOwner        = hwnd ;
     fr.hInstance        = NULL ;
     fr.Flags            = FR_HIDEUPDOWN | FR_HIDEMATCHCASE | FR_HIDEWHOLEWORD ;
     fr.lpstrFindWhat    = szFindText ;
     fr.lpstrReplaceWith = NULL ;
     fr.wFindWhatLen     = MAX_STRING_LEN ;
     fr.wReplaceWithLen  = 0 ;
```

```
     fr.lCustData          = 0 ;
     fr.lpfnHook           = NULL ;
     fr.lpTemplateName     = NULL ;

     return FindText (&fr) ;
}

HWND PopFindReplaceDlg (HWND hwnd)
{
     static FINDREPLACE fr ;       // must be static for modeless dialog!!!

     fr.lStructSize        = sizeof (FINDREPLACE) ;
     fr.hwndOwner          = hwnd ;
     fr.hInstance          = NULL ;
     fr.Flags              = FR_HIDEUPDOWN | FR_HIDEMATCHCASE | FR_HIDEWHOLEWORD ;
     fr.lpstrFindWhat      = szFindText ;
     fr.lpstrReplaceWith   = szReplText ;
     fr.wFindWhatLen       = MAX_STRING_LEN ;
     fr.wReplaceWithLen    = MAX_STRING_LEN ;
     fr.lCustData          = 0 ;
     fr.lpfnHook           = NULL ;
     fr.lpTemplateName     = NULL ;

     return ReplaceText (&fr) ;
}

BOOL PopFindFindText (HWND hwndEdit, int * piSearchOffset, LPFINDREPLACE pfr)
{
     int    iLength, iPos ;
     PTSTR  pstrDoc, pstrPos ;

          // Read in the edit document

     iLength = GetWindowTextLength (hwndEdit) ;

     if (NULL == (pstrDoc = (PTSTR) malloc ((iLength + 1) * sizeof (TCHAR))))
          return FALSE ;

     GetWindowText (hwndEdit, pstrDoc, iLength + 1) ;

          // Search the document for the find string

     pstrPos = _tcsstr (pstrDoc + * piSearchOffset, pfr->lpstrFindWhat) ;
     free (pstrDoc) ;

          // Return an error code if the string cannot be found

     if (pstrPos == NULL)
```

```
            return FALSE ;

              // Find the position in the document and the new start offset

     iPos = pstrPos - pstrDoc ;
     * piSearchOffset = iPos + lstrlen (pfr -> lpstrFindWhat) ;

              // Select the found text

     SendMessage (hwndEdit, EM_SETSEL, iPos, * piSearchOffset) ;
     SendMessage (hwndEdit, EM_SCROLLCARET, 0, 0) ;

     return TRUE ;
}

BOOL PopFindNextText (HWND hwndEdit, int * piSearchOffset)
{
     FINDREPLACE fr ;

     fr.lpstrFindWhat = szFindText ;

     return PopFindFindText (hwndEdit, piSearchOffset, &fr) ;
}

BOOL PopFindReplaceText (HWND hwndEdit, int * piSearchOffset, LPFINDREPLACE pfr)
{
            // Find the text

     if (!PopFindFindText (hwndEdit, piSearchOffset, pfr))
          return FALSE ;

            // Replace it

     SendMessage (hwndEdit, EM_REPLACESEL, 0, (LPARAM) pfr -> lpstrReplaceWith) ;

     return TRUE ;
}

BOOL PopFindValidFind (void)
{
     return * szFindText != '\0' ;
}
```

POPFONT.C

```
/* ---------------------
     POPFONT.C——Popup Editor Font Functions
   ----------------------*/
```

```
#include <windows.h>
#include <commdlg.h>

static LOGFONT  logfont ;
static HFONT    hFont ;

BOOL PopFontChooseFont (HWND hwnd)
{
     CHOOSEFONT cf ;
     cf.lStructSize      = sizeof (CHOOSEFONT) ;
     cf.hwndOwner        = hwnd ;
     cf.hDC              = NULL ;
     cf.lpLogFont        = &logfont ;
     cf.iPointSize       = 0 ;
     cf.Flags            = CF_INITTOLOGFONTSTRUCT | CF_SCREENFONTS | CF_EFFECTS ;
     cf.rgbColors        = 0 ;
     cf.lCustData        = 0 ;
     cf.lpfnHook         = NULL ;
     cf.lpTemplateName   = NULL ;
     cf.hInstance        = NULL ;
     cf.lpszStyle        = NULL ;
     cf.nFontType        = 0 ;          // Returned from ChooseFont
     cf.nSizeMin         = 0 ;
     cf.nSizeMax         = 0 ;

     return ChooseFont (&cf) ;
}

void PopFontInitialize (HWND hwndEdit)
{
     GetObject (GetStockObject (SYSTEM_FONT), sizeof (LOGFONT),
                (PTSTR) &logfont) ;

     hFont = CreateFontIndirect (&logfont) ;
     SendMessage (hwndEdit, WM_SETFONT, (WPARAM) hFont, 0) ;
}

void PopFontSetFont (HWND hwndEdit)
{
     HFONT hFontNew ;
     RECT  rect ;

     hFontNew = CreateFontIndirect (&logfont) ;
     SendMessage (hwndEdit, WM_SETFONT, (WPARAM) hFontNew, 0) ;
     DeleteObject (hFont) ;
     hFont = hFontNew ;
     GetClientRect (hwndEdit, &rect) ;
     InvalidateRect (hwndEdit, &rect, TRUE) ;
```

```
}

void PopFontDeinitialize (void)
{
     DeleteObject (hFont) ;
}
```

POPPRNT0.C

```
/*---------------------------------------------------------------
     POPPRNT0.C -- Popup Editor Printing Functions (dummy version)
  ---------------------------------------------------------------*/

#include <windows.h>

BOOL PopPrntPrintFile (HINSTANCE hInst, HWND hwnd, HWND hwndEdit,
                       PTSTR pstrTitleName)
{
     return FALSE ;
}
```

POPPAD.RC（摘录）

```
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
// Dialog

ABOUTBOX DIALOG DISCARDABLE 32, 32, 180, 100
STYLE DS_MODALFRAME | WS_POPUP
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON   "OK",IDOK,66,80,50,14
    ICON            "POPPAD",IDC_STATIC,7,7,20,20
    CTEXT           "PopPad",IDC_STATIC,40,12,100,8
    CTEXT           "Popup Editor for Windows",IDC_STATIC,7,40,166,8
    CTEXT           "(c) Charles Petzold, 1998",IDC_STATIC,7,52,166,8
END

PRINTDLGBOX DIALOG DISCARDABLE 32, 32, 186, 95
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "PopPad"
FONT 8, "MS Sans Serif"
BEGIN
    PUSHBUTTON      "Cancel",IDCANCEL,67,74,50,14
    CTEXT           "Sending",IDC_STATIC,8,8,172,8
```

```
    CTEXT           "",IDC_FILENAME,8,28,172,8
    CTEXT           "to print spooler.",IDC_STATIC,8,48,172,8
END

//////////////////////////////////////////////////////////////////////////////
// Menu

POPPAD MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New \ tCtrl + N ",           IDM_FILE_NEW
        MENUITEM "&Open... \ tCtrl + O ",       IDM_FILE_OPEN
        MENUITEM "&Save \ tCtrl + S",           IDM_FILE_SAVE
        MENUITEM "Save &As...",                 IDM_FILE_SAVE_AS
        MENUITEM SEPARATOR
        MENUITEM "&Print \ tCtrl + P ",         IDM_FILE_PRINT
        MENUITEM SEPARATOR
        MENUITEM "E&xit ",                      IDM_APP_EXIT
    END
    POPUP "&Edit"
    BEGIN
        MENUITEM "&Undo \ tCtrl + Z",           IDM_EDIT_UNDO
        MENUITEM SEPARATOR
        MENUITEM "Cu&t \ tCtrl + X",            IDM_EDIT_CUT
        MENUITEM "&Copy \ tCtrl + C",           IDM_EDIT_COPY
        MENUITEM "&Paste \ tCtrl + V ",         IDM_EDIT_PASTE
        MENUITEM "De&lete \ tDel ",             IDM_EDIT_CLEAR
        MENUITEM SEPARATOR
        MENUITEM "&Select All ",                IDM_EDIT_SELECT_ALL
    END
    POPUP "&Search"
    BEGIN
        MENUITEM "&Find... \ tCtrl + F ",       IDM_SEARCH_FIND
        MENUITEM "Find &Next \ tF3",            IDM_SEARCH_NEXT
        MENUITEM "&Replace... \ tCtrl + R ",    IDM_SEARCH_REPLACE
    END
    POPUP "F&ormat "
    BEGIN
        MENUITEM "&Font...",                    IDM_FORMAT_FONT
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&Help",                       IDM_HELP
        MENUITEM "&About PopPad...",            IDM_APP_ABOUT
    END
```

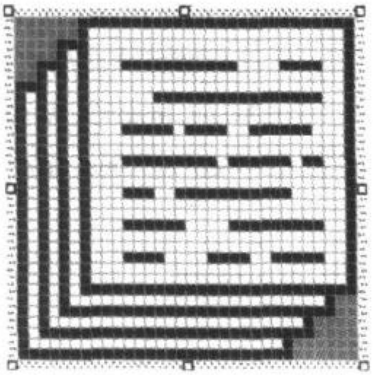
```
END
/////////////////////////////////////////////////////////////////////////////
// Accelerator

POPPAD ACCELERATORS DISCARDABLE
BEGIN
    VK_BACK,        IDM_EDIT_UNDO,          VIRTKEY, ALT, NOINVERT
    VK_DELETE,      IDM_EDIT_CLEAR,         VIRTKEY, NOINVERT
    VK_DELETE,      IDM_EDIT_CUT,           VIRTKEY, SHIFT, NOINVERT
    VK_F1,          IDM_HELP,               VIRTKEY, NOINVERT
    VK_F3,          IDM_SEARCH_NEXT,        VIRTKEY, NOINVERT
    VK_INSERT,      IDM_EDIT_COPY,          VIRTKEY, CONTROL, NOINVERT
    VK_INSERT,      IDM_EDIT_PASTE,         VIRTKEY, SHIFT, NOINVERT
    "^C",           IDM_EDIT_COPY,          ASCII, NOINVERT
    "^F",           IDM_SEARCH_FIND,        ASCII, NOINVERT
    "^N",           IDM_FILE_NEW,           ASCII, NOINVERT
    "^O",           IDM_FILE_OPEN,          ASCII, NOINVERT
    "^P",           IDM_FILE_PRINT,         ASCII, NOINVERT
    "^R",           IDM_SEARCH_REPLACE,     ASCII, NOINVERT
    "^S",           IDM_FILE_SAVE,          ASCII, NOINVERT
    "^V",           IDM_EDIT_PASTE,         ASCII, NOINVERT
    "^X",           IDM_EDIT_CUT,           ASCII, NOINVERT
    "^Z",           IDM_EDIT_UNDO,          ASCII, NOINVERT
END

/////////////////////////////////////////////////////////////////////////////
// Icon

POPPAD              ICON    DISCARDABLE     "poppad.ico"
```

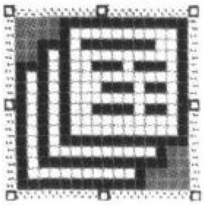
RESOURCE.H（摘录）

```
// Microsoft Developer Studio generated include file.
// Used by poppad.rc

#define IDC_FILENAME            1000
#define IDM_FILE_NEW            40001
#define IDM_FILE_OPEN           40002
#define IDM_FILE_SAVE           40003
#define IDM_FILE_SAVE_AS        40004
#define IDM_FILE_PRINT          40005
#define IDM_APP_EXIT            40006
#define IDM_EDIT_UNDO           40007
#define IDM_EDIT_CUT            40008
#define IDM_EDIT_COPY           40009
#define IDM_EDIT_PASTE          40010
```

```
#define IDM_EDIT_CLEAR          40011
#define IDM_EDIT_SELECT_ALL     40012
#define IDM_SEARCH_FIND         40013
#define IDM_SEARCH_NEXT         40014
#define IDM_SEARCH_REPLACE      40015
#define IDM_FORMAT_FONT         40016
#define IDM_HELP                40017
#define IDM_APP_ABOUT           40018
```

POPPAD.ICO





为了避免在第十三章中重复源代码，我在 POPPAD.RC 的菜单中添加了打印项目和一些其他的支持。

POPPAD.C 包含了程序中所有的基本源代码。POPFILE.C 具有激活 File Open 和 File Save 对话框的代码，它还包含文件 I/O 例程。POPFIND.C 中包含了查找和替换逻辑。POPFONT.C 包含了字体选择逻辑。POPPRNT0.C 不完成什么工作：在第十三章中将使用 POPPRNT.C 替换 POPPRNT0.C，以创建最终的 POPPAD 程序。

让我们先来看一看 POPPAD.C。POPPAD.C 含有两个文件名串：第一个保存在 WndProc，名称为 szFileName，含有详细的驱动器名、路径名和文件名；第二个保存为 szTitleName，是程序本身的文件名。它用在 POPPAD3 的 DoCaption 函数中，以便将文件名显示在窗口的标题栏上；也用在 OKMessage 函数和 AskAboutSave 函数中，以便向用户显示消息框。

POPFILE.C 包含了几个显示 File Open 和 File Save 对话框以及实际执行文件 I/O 的函数。对话框是使用函数 GetOpenFileName 和 GetSaveFileName 来显示的，这两个函数都使用一个类型为 OPENFILENAME 的结构，这个结构在 COMMDLG.H 中定义。在 POPFILE.C 中，使用了一个该结构类型的全局变量，取名为 ofn。ofn 的大多数字段在 PopFileInitialize 函数中被初始化，POPPAD.C 在 WndProc 中处理 WM_CREATE 消息时调用该函数。

将 ofn 作为静态全局结构变量比较方便，因为 GetOpenFileName 和 GetSaveFileName 给该结构返回的一些信息将在以后调用这些函数时用到。

尽管通用对话框具有许多选项——包括设置自己的对话框模板，以及为对话框过程增加“钩子(hook)”——POPFILE.C 中使用的 File Open 和 File Save 对话框是最基本的。OPENFILENAME 结构中被设置的字段只有 lStructSize(结构的长度)、hwndOwner(对话框拥有者)、lpstrFilter(下面将简要讨论)、lpstrFile 和 nMaxFile(指向接收完整文件名的缓冲区指针和该缓冲区的大小)，以及 lpstrFileTitle 和 nMaxFileTitle(文件名缓冲区及其大小)、Flags(设置对话框的选项)和 lpstrDefExt(如果用户在对话框中输入文件名时不指定文件扩展名，那么它就是默认的文件扩展名)。

当用户在 File 菜单中选择 Open 时，POPPAD3 调用 POPFILE 的 PopFileOpenDlg 函数，将窗口句柄、一个指向文件名缓冲区的指针和一个指向文件标题缓冲区的指针传给它。PopFileOpenDlg 恰当地设置 OPENFILENAME 结构的 hwndOwner、lpstrFile 和 lpstrFileTitle 字段，将 Flags 设置为 OFN_CREATEPROMPT，然后调用 GetOpenFileName，显示如图 11-6 所示的普通对话框。

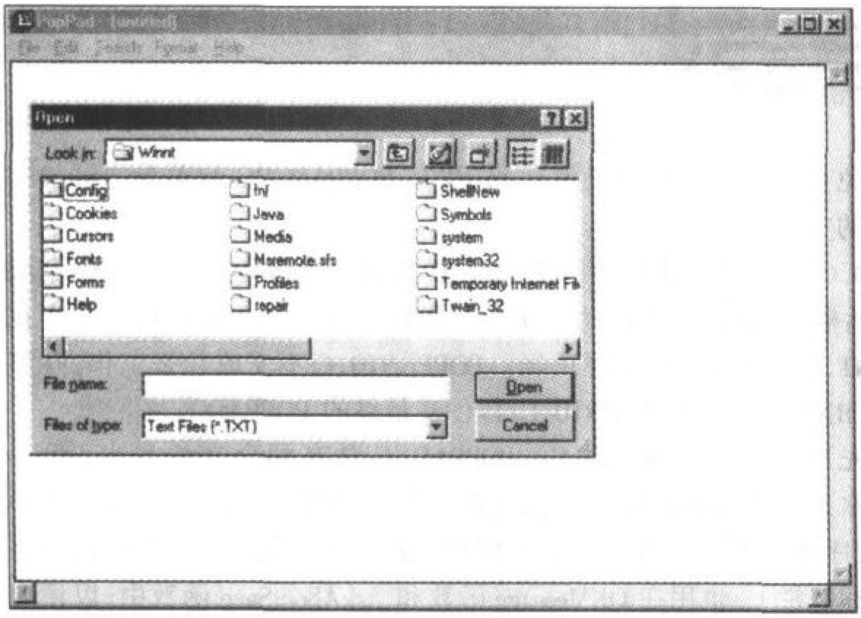


图 11-6 File Open 对话框

当用户结束这个对话框时，GetOpenFileName 函数返回。OFN_CREATEPROMPT 标志指示 GetOpenFileName 显示一个消息框，询问用户如果所选文件不存在，是否创建该文件。

左下角的组合框列出了将要显示在文件列表中的文件类型，此列表框被称为“过滤

器”。用户可以通过从组合框列表中选择一种不同的文件类型来更改过滤器。在 POPFILE.C 的 PopFileInitialize 函数中，我在变量 szFilter（一个字符串数组）中为三种类型的文件定义了一个过滤器：带有 .TXT 扩展名的文本文件、带有 .ASC 扩展名的 ASCII 文件，以及所有文件。该数组的第一个字符串的指针被设置为指向 OPENFILENAME 结构的 lpstrFilter 字段。

如果用户在对话框处于活动状态时更改了过滤器，那么 OPENFILENAME 的 nFilterIndex 字段反映出用户的选择。由于该结构是静态变量，下次激活对话框时，过滤器将被设置为选中的文件类型。

POPFILE.C 中的 PopFileSaveDlg 函数与此类似，它将 Flags 参数设置为 OFN_OVERWRITEPROMPT，并调用 GetSaveFileName 函数以激活 File Save 对话框。OFN_OVERWRITEPROMPT 标志导致显示一个消息框，如果选中的文件已经存在，那么将询问用户是否覆盖该文件。

11.3.2 Unicode 文件 I/O

对于本书中的大多数程序，您都不必注意 Unicode 和非 Unicode 版的区别。例如，在 POPPAD3 的 Unicode 中，编辑控制将保留 Unicode 文本和使用 Unicode 文本串的所有通用对话框。例如，当程序需要查找和替换时，所有的操作都处理 Unicode 串，而不需要转换。

不过，POPPAD3 处理文件 I/O，也就是说，程序不会自己附带自己。如果 Unicode 版的 POPPAD3 获得了编辑缓冲区的内容并将其写入了磁盘，文件将是 Unicode。如果非 Unicode 版的 POPPAD3 读取了该文件，并将其写入编辑缓冲区，其结果将是一堆垃圾。Unicode 版读取由非 Unicode 版保存的文件时也会这样。

解决的办法在于辨别和转换。首先，在 POPFILE.C 的 PopFileWrite 函数中，您将看到 Unicode 版的程序将在文件的开始位置写入 0xFEFF。这定义为字节顺序标记，以表明文本文件含有 Unicode 文本。

其次，在 PopFileRead 函数中，程序用 IsTextUnicode 函数来决定文件是否含有字节顺序标记。此函数甚至检测字节顺序标记是否被反转，即 Unicode 文本文件是在 Macintosh 或者其他使用与 Intel 处理器相反的字节顺序的机器上创建的。这时，字节的每一部分都将被反转。如果文件是 Unicode 版，但是被非 Unicode 版的 POPPAD3 读取，那么这时，文本将被 WideCharToMultiChar 转换，WideCharToMultiChar 实际上是一个宽字符 ANSI 函数（除非运行的是远东版的 Windows）。只有这时文本才可以被放置到编辑缓冲区中。

类似地，如果文件是非 Unicode 文本文件，而运行的是 Unicode 版的程序，那么文本必须用 MultiCharToWideChar 转换。

11.3.3 更改字体

我们将在第十七章详细讨论字体，但那些都不能代替通用对话框函数来选择字体。

在 WM_CREATE 消息期间，POPFONT.C 中的 POPPAD 调用 PopFontInitialize。这个函数获取一个基于系统字体的 LOGFONT 结构，由此创建一种字体，并向编辑控制发送一个 WM_SETFONT 消息来设置一种新的字体。（默认编辑控制字体是系统字体，而 PopFontInitialize 为编辑控制创建一种新的字体，因为最终该字体将被删除，而删除现有系统字体是不明智的。）

当 POPPAD 收到来自程序的字体选项的 WM_COMMAND 消息时，它调用 PopFont-ChooseFont。这个函数初始化一个 CHOOSEFONT 结构，然后调用 ChooseFont 显示字体选择对话框。如果用户按下 OK 按钮，那么 ChooseFont 将返回 TRUE。随后，POPPAD 调用 PopFontSetFont 来设置编辑控制中的新字体，旧字体将被删除。

最后，在 WM_DESTROY 消息期间，POPPAD 调用 PopFontDeinitialize 来删除最近一次由 PopFontSetFont 创建的字体。

11.3.4 查找与替换

通用对话框库也包含两个用于文本查找和替换函数的对话框，这两个函数（FindText 和 ReplaceText）使用一个类型为 FINDREPLACE 的结构。程序 10-9 中所示的 POPFIND.C 文件有两个例程（PopFindFindDlg 和 PopFindReplaceDlg）调用这些函数，还有两个函数在编辑控制中查找和替换文本。

使用查找和替换函数有一些考虑。首先，它们激活的对话框是非模态对话框，这意味着必须改写消息循环，以便在对话框活动时调用 IsDialogMessage。第二，传送给 Find-Text 和 ReplaceText 的 FINDREPLACE 结构必须是一个静态变量，因为对话框是模态的，函数在对话框显示之后返回，而不是在对话框结束之后返回；而对话框过程必须仍然能够访问该结构。

第三，在显示 FindText 和 ReplaceText 对话框时，它们通过一个特殊消息与拥有者窗口进行通讯，消息号可以通过以 FINDMSGSTRING 为参数调用 RegisterWindowMessage 函数来获得。这是在 WndProc 中处理 WM_CREATE 消息时完成的，消息号存放在静态变量中。

在处理默认消息时，WndProc 将消息变量与 RegisterWindowMessage 返回的值相比较。lParam 消息参数是一个指向 FINDREPLACE 结构的指针，Flags 字段指示用户使用对话框是为了查找文本还是替换文本，以及是否要终止对话框。POPPAD3 调用 POPFIND.C 中的 PopFindFindText 和 PopFindReplaceText 函数来执行查找和替换功能。

11.3.5 只调用一个函数的 Windows 程序

到现在为止,我们已经给出了两个程序让您浏览选定的颜色,它们分别是第九章中的 COLORS1 和本章中的 COLORS2。现在是给出 COLORS3 的时候了,这个程序只有一个 Windows 函数调用。COLORS3 的源代码如程序 11-7 所示。

程序 11-7 COLORS3 程序

COLORS3.C

```
/*-------------------------------------------------------
   COLORS3.C——Version using Common Dialog Box
              (c) Charles Petzold, 1998
  -------------------------------------------------------*/

#include <windows.h>
#include <commdlg.h>

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static CHOOSECOLOR cc ;
     static COLORREF    crCustColors[16] ;

     cc.lStructSize    = sizeof (CHOOSECOLOR) ;
     cc.hwndOwner      = NULL ;
     cc.hInstance      = NULL ;
     cc.rgbResult      = RGB (0x80, 0x80, 0x80) ;
     cc.lpCustColors   = crCustColors ;
     cc.Flags          = CC_RGBINIT | CC_FULLOPEN ;
     cc.lCustData      = 0 ;
     cc.lpfnHook       = NULL ;
     cc.lpTemplateName = NULL ;

     return ChooseColor (&cc) ;
}
```

COLORS3 调用的唯一个 Windows 函数是 ChooseColor,这也是通用对话框库中的函数,它显示如图 11-7 所示的对话框。颜色选择类似于 COLORS1 和 COLORS2,但是它的交互能力更强。

ChooseColor 函数使用一个 CHOOSECOLOR 类型的结构和含有 16 个 DWORD 的数组来存储自定义的颜色,用户将从对话框中选择其中一种颜色。rgbResult 字段可以初始化为一个颜色值,如果 Flags 字段的 CC_RGBINIT 标志被置位,则显示该颜色。通常在

使用这个函数时，rgbResult 将被设置为用户选择的颜色。

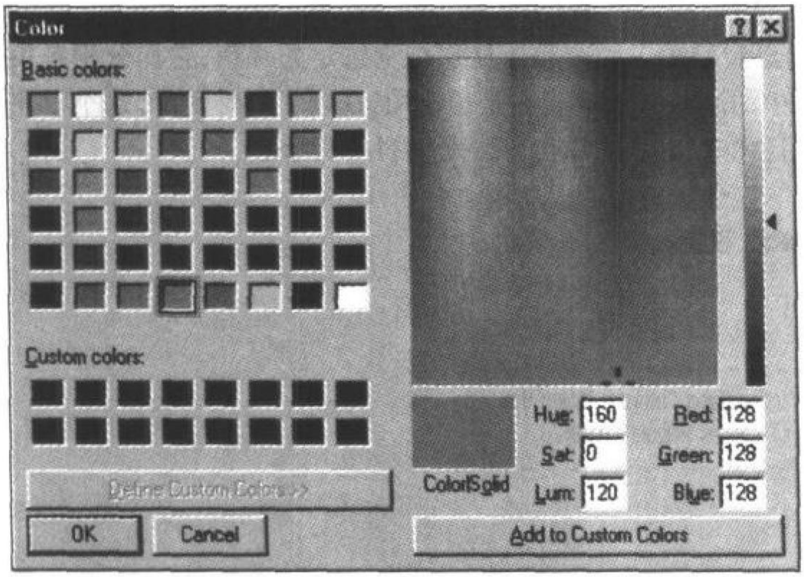


图 11-7　COLORS3 的屏幕显示

注意，Color 对话框的 hwndOwner 字段被设置为 NULL；在 ChooseColor 函数调用 DialogBox 以显示对话框时，DialogBox 的第三个参数也被设置为 NULL。这是完全合法的，其含义是对话框不为另一个窗口所拥有。对话框的标题将显示在任务栏中，而对话框就像一个普通的窗口那样运行。

您也可以在自己程序的对话框中使用这种技巧，使 Windows 程序只创建对话框，其他事情都在对话框过程中完成，这是可能的。

第十二章

剪贴板

Microsoft Windows 剪贴板允许把数据从一个程序传送到另一个程序。它的原理相对而言比较简单,把数据存放到剪贴板上的程序或从剪贴板上取出数据的程序都无需太多的开销。Windows 98 和 Microsoft Windows NT 都提供了剪贴板查看程序,该程序可以显示剪贴板的当前内容。

许多处理文件或者其他数据的程序都包含一个 Edit 菜单,其中包括 Cut、Copy 和 Paste 选项。当用户选择 Cut 或者 Copy 时,程序将数据传送到剪贴板。这个数据使用某种格式,如文本、位图(一种按位排列的矩形数组,其中的位与显示表面的像素相对应),或者元文件(用二进制表示的绘图命令集)等。当用户从菜单中选择 Paste 时,程序检查剪贴板中包含的数据使用的是否是程序可以接受的格式。如果是,那么数据将从剪贴板传送到程序中。

如果用户不发出明确的指令,程序就不能把数据送入或移出剪贴板。例如,在某个程序中执行剪切或复制(Ctrl-X 或 Ctrl-C)操作的用户,应该能够假定数据将保存在剪贴板上,直到下次剪切或复制操作为止。

第十和第十一章所示的 POPPAD 程序的修订版中,您可以再次调用 Edit 菜单。但是,编辑控制中只包括了发送消息。多数情况下,并不方便,您必须自己替换调用剪贴板传输函数。

本章集中讨论将文本传入和移出剪贴板。在后面的章节里,我将向您展示如何用剪贴板处理位图(第十四、十五和十六章)和元文件(第十八章)。

12.1 剪贴板的简单使用

我们从分析将数据传送到剪贴板(剪切或复制)和访问剪贴板数据(粘贴)的代码开始。

12.1.1 标准剪贴板数据格式

Windows 支持不同的预先定义的剪贴板格式，这些格式在 WINUSER.H 定义成以 CF 为前缀的标识符。

首先介绍三种能保存在剪贴板上的文本数据类型，以及一种与剪贴板格式相关的数据类型：

- **CF_TEXT** 以 NULL 结尾的 ANSI 字符集字符串，它在每行末尾包含一个回车和换行符，这是最简单的剪贴板数据格式。传送到剪贴板的数据存放在全局内存块中，并且是利用内存块句柄进行传送的（我将简要地介绍此概念）。这个内存块专供剪贴板使用，创建它的程序不应继续使用它。
- **CF_OEMTEXT** 含有文本数据（与 CF_TEXT 类似）的内存块，但是它使用的是 OEM 字符集。通常 Windows 程序不必关心这一点；它只有与在窗口中运行的 MS-DOS 程序一起使用剪贴板时才使用。
- **CF_UNICODETEXT** 含有 Unicode 文本的内存块。与 CF_TEXT 类似，它在每一行的末尾包含一个回车和换行符，以及一个 NULL 字符（两个 0 字节），以表示数据结束。CF_UNICODETEXT 只支持 Windows NT。
- **CF_LOCALE** 一个本地标识符句柄，表明本地访问剪贴板文本。

下面是两种附加的剪贴板格式，它们在概念上与 CF_TEXT 格式相似（也就是说，它们也是基于文本的），但是它们不需要以 NULL 结尾，因为格式已经定义了数据的结尾。现在已经很少使用这些格式了：

- **CF_SYLK** 包含 Microsoft “符号链接”数据格式的全局内存块。这种格式在 Microsoft 的 Multiplan、Chart 和 Excel 程序之间交换数据，它是一种 ASCII 码格式，每一行都以回车和换行结尾。
- **CF_DIF** 包含数据交换格式（DIF）的数据的全局内存块。这种格式是由 Software Arts 公司提出的，用于把数据传输到 VisiCalc 电子表格程序中。这也是一种 ASCII 码格式，每一行都以回车和换行结尾。

下面三种剪贴板格式用于连结位图，所谓位图就是数据位的矩形数组，其中的数据位与输出设备的像素相对应。第十四和第十五章将详细讨论位图及这些位图的剪贴板格式：

- **CF_BITMAP** 与设备有关的位图格式。位图是通过位图句柄传送给剪贴板的。同样，在把这个位图传送给剪贴板之后，程序不应再继续使用这个位图。
- **CF_DIB** 定义一个设备无关位图的内存块（在第十五章中描述）。这种内存块是

以位图信息结构开始的,后跟可用的颜色表和位图数据位。

- **CF_PALETTE** 调色板句柄。它通常与 CF_DIB 配合使用,以定义与设备有关的位图所使用的颜色调色板。

在剪贴板中,还有可能以工业标准的 TIFF 格式保存的位图数据:

- **CF_TIFF** 含有标记图像文件格式(TIFF)数据的全局内存块。这种格式由 Microsoft、Aldus 公司和 Hewlett-Packard 公司,以及一些硬件厂商推荐使用。这一格式可从 Hewlett-Packard 的 Web 站点获得。

下面是两个元文件格式,我将在第十八章详细讨论。一个元文件就是一个以二进制格式保存的绘图命令集:

- **CF_METAFILEPICT** 基于支持 Windows 的老的元文件的“元文件图片”。
- **CF_ENHMETAFILE** 增强型元文件(支持 32 位的 Windows)的句柄。

最后介绍几个混合型的剪贴板格式:

- **CF_PENDATA** 与 Windows 的画笔扩展联合使用;
- **CF_WAVE** 声音(波形)文件;
- **CF_RIFF** 使用资源交换文件格式的多媒体数据;
- **CF_HDROP** 与拖放服务相关的文件列表。

12.1.2 内存分配

程序向剪贴板传输数据的时候,必须分配一个内存块,并且剪贴板将在根本上拥有它。在本书前面的程序中需要分配内存时,只需使用标准 C 运行库所支持的 malloc 函数。但是,由于在 Windows 中运行的应用程序之间必须要共享剪贴板所保存的内存块,这时 malloc 函数就有些不适应此项任务了。

实际上,我们必须挖掘为早期的 Windows 所开发的内存分配函数,那时的操作系统运行在 16 位的实模式内存结构中。现在仍然支持这些函数,您还可以使用它们,但不是必须使用。

要用 Windows API 来分配一个内存块,可以调用

```
hGlobal = GlobalAlloc (uiFlags, dwSize) ;
```

此函数有两个参数:一系列可能的标志和内存块的字节大小。它返回一个 HGLOBAL 类型的句柄,称为“全局内存块句柄”或“全局句柄”。返回值为 NULL 表示不能分配足够的内存。

虽然 GlobalAlloc 的两个参数略有一点不同,但它们都是 32 位的无符号整数。如果

将第一个参数设置为 0，那么就可以更有效地使用标志 GMEM_FIXED。这种情况下，GlobalAlloc 返回的全局句柄实际是指向分配的内存块的指针。

如果不喜欢将内存块中的每一位都初始化为 0，那么您也可以使用标志 GMEM_ZEROINIT。在 Windows 头文件中，紧凑的 GPTR 标志定义为 GMEM_FIXED 和 GMEM_ZEROINIT 标志的组合：

```
# define GPTR (GMEM_FIXED | GMEM_ZEROINIT)
```

下面是一个再分配函数：

```
hGlobal = GlobalReAlloc (hGlobal, dwSize, uiFlags) ;
```

如果内存块扩大了，您可以用 GMEM_ZEROINIT 标志将新的字节置 0。

下面是获得内存块大小的函数：

```
dwSize = GlobalSize (hGlobal) ;
```

和释放内存块的函数：

```
GlobalFree (hGlobal) ;
```

在早期 16 位的 Windows 中，因为 Windows 不能在物理内存中移动内存块，所以禁止使用 GMEM_FIXED 标志。在 32 位的 Windows 中，GMEM_FIXED 标志很普通，这是因为它将返回一个虚拟地址，并且操作系统也能够通过更改页表来在物理内存中移动内存块。为 16 位的 Windows 编程时，实际上在 GlobalAlloc 中推荐使用 GMEM_MOVEABLE 标志。在 Windows 的头文件中还定义了一个速记标识符，用此标识符可以在可移动的内存之外填 0：

```
# define GHND (GMEM_MOVEABLE | GMEM_ZEROINIT)
```

GMEM_MOVEABLE 标志允许 Windows 在虚拟内存中移动一个内存块。这不是说将在物理内存中移动内存块，只是应用程序用于读写的块地址发生了变化。

尽管 GMEM_MOVEABLE 是 16 位 Windows 的规则，但是它现在还是很少使用了。然而，如果您的应用程序频繁地分配、再分配，以及释放不同大小的内存块，那么应用程序的虚拟地址空间将会变得支离破碎。幸运的是，这时您可以在虚拟内存地址上运行。如果这是一个潜在的问题，那么您将希望使用可移动内存。下面就介绍如何做到这一点。

首先定义一个指针（例如，一个 int 型指针）和一个 GLOBALHANDLE 类型的变量：

```
int * p ;
GLOBALHANDLE hGlobal ;
```

然后分配内存。例如：

```
hGlobal = GlobalAlloc (GHND, 1024) ;
```

与处理其他 Windows 句柄一样,您不必过分地关心数字的实际意义,只需保存就可以了。需要访问内存块时,可以调用

```
p = (int * ) GlobalLock (hGlobal) ;
```

此函数将句柄转换为指针。在块锁定期间,Windows 将固定虚拟内存中的地址,不再移动。访问结束后调用

```
GlobalUnlock (hGlobal) ;
```

这将使 Windows 可以在虚拟内存中移动内存块。要真正地强制执行此过程(也是早期 Windows 程序员的痛苦经历),您应该在一个单独的消息期间锁定和解锁内存块。

在释放内存时,调用 GlobalFree 应使用句柄而不是指针。如果您现在不能访问句柄,可以使用下面的函数:

```
hGlobal = GlobalHandle (p) ;
```

在解锁之前,您可以多次锁定一个内存块。Windows 保留一个锁定数,而且在内存块自由移动之前,每次锁定都需要相应的解锁。当 Windows 在虚拟内存中移动一个内存块时,不需要将字节从一个位置复制到另一个,只需巧妙地处理页表。通常,32 位 Windows 为程序分配可移动的内存块实际是为了防止虚拟内存的破碎。使用剪贴板时,也应该使用可移动内存。

为剪贴板分配内存时,您应该使用带有 GMEM_MOVEABLE 和 GMEM_MOVEABLE 标志的 GlobalAlloc 函数。GMEM_SHARE 标志使其他应用程序可以使用内存块。

12.1.3 将文本传送到剪贴板

让我们假想把一个 ANSI 字符串传送到剪贴板上,并且我们已经有了指向这个字符串的指针(pString)。现在希望传送这个字符串的 iLength 字符,这些字符可能以 NULL 结尾,也可能不以 NULL 结尾。

首先,通过使用 GlobalAlloc 来分配一个足以保存字符串的内存块,其中还包括一个终止符 NULL:

```
hGlobal = GlobalAlloc (GHND | GMEM_SHARE, iLength + 1) ;
```

如果未能分配到内存块,hGlobal 的值将为 NULL 。如果分配成功,则锁定这个块,并得到指向它的一个指针:

```
pGlobal = GlobalLock (hGlobal) ;
```

将字符串复制到内存块中:

```
for (i = 0 ; i < wLength ; i++ )
    * pGlobal ++ = * pString ++ ;
```

由于 GlobalAlloc 的 GHND 标志已使整个内存块在分配期间被清零,所以不需要增加终止符 NULL 。以下语句为内存块解锁:

```
GlobalUnlock (hGlobal) ;
```

现在就有了引用以 NULL 结尾的文本所在内存块的内存句柄。要将它传送到剪贴板中,打开剪贴板并把它清空:

```
OpenClipboard (hwnd) ;
EmptyClipboard () ;
```

利用 CF_TEXT 标识符把内存句柄交给剪贴板,再关闭剪贴板:

```
SetClipboardData (CF_TEXT, hGlobal) ;
CloseClipboard () ;
```

工作告一段落。

下面是关于此过程的一些规则:

- 在处理同一个消息的过程中调用 OpenClipboard 和 CloseClipboard。不需要时,不要打开剪贴板。
- 不要把锁定的内存句柄交给剪贴板。
- 当调用 SetClipboardData 后,不要再继续使用该内存块。它不再属于用户程序,必须把句柄看成是无效的。如果需要继续访问数据,可以制作数据的副本,或从剪贴板中读取它(如下节所述)。您也可以在 SetClipboardData 调用和 CloseClipboard 调用之间继续引用内存块,但是不要使用传递给 SetClipboardData 函数的全局句柄。事实上,此函数也返回一个全局句柄,必需锁定这些句柄以访问内存。在调用 CloseClipboard 之前,应先为此句柄解锁。

12.1.4 从剪贴板上获取文本

从剪贴板上获取文本只比把文本传送到剪贴板上稍微复杂一些。您必须首先确定剪贴板是否含有 CF_TEXT 格式的数据,最简单的方法是调用

```
bAvailable = IsClipboardFormatAvailable (CF_TEXT) ;
```

如果剪贴板上含有 CF_TEXT 数据,这个函数将返回 TRUE(非 0)。我们在第十章的 POPPAD2 程序中已使用了这个函数,用它来确定 Edit 菜单中 Paste 项是被启用的还是被灰化的。IsClipboardFormatAvailable 是少数几个不需要首先打开剪贴板就可以使用的剪贴板函数之一。但是,如果您以后再打开剪贴板得到这个文本,应该再做一次检查

(使用同样的函数或其他方法),以便确定 CF_TEXT 数据是否仍然留在剪贴板中。

为了传送出文本,首先打开剪贴板:

```
OpenClipboard (hwnd) ;
```

得到引用文本的内存块句柄:

```
hGlobal = GetClipboardData (CF_TEXT) ;
```

如果剪贴板不包含 CF_TEXT 格式的数据,此句柄就为 NULL。这是确定剪贴板是否含有文本的另一种方法。如果 GetClipboardData 返回 NULL,则关闭剪贴板,不能做其他任何工作。

从 GetClipboardData 得到的句柄并不属于用户程序——它属于剪贴板。仅在 GetClipboardData 和 CloseClipboard 调用之间这个句柄才有效。您不能释放这个句柄或更改它所引用的数据。如果需要继续访问这些数据,必须制作这个内存块的副本。

这里有一种将数据复制到用户程序中的方法。首先,分配一块与剪贴板数据块大小相同的内存块,并分配一个指向该块的指针:

```
pText = (char * ) malloc (GlobalSize (hGlobal)) ;
```

再次调用 hGlobal ,而 hGlobal 是从 GetClipboardData 调用返回的全局句柄。现在锁定句柄,获得一个指向剪贴板块的指针:

```
pGlobal = GlobalLock (hGlobal) ;
```

现在就可以复制数据了:

```
strcpy (pText, pGlobal) ;
```

或者,您可以使用一些简单的 C 代码:

```
while ( * pText ++ = * pGlobal ++ ) ;
```

在关闭剪贴板之前先解锁内存块:

```
GlobalUnlock (hGlobal) ;
CloseClipboard () ;
```

现在有了一个叫做 pText 的指针,以后程序的用户就可以用它来复制文本了。

12.1.5 打开和关闭剪贴板

在任何时候,只有一个程序可以打开剪贴板。调用 OpenClipboard 的作用是当一个程序使用剪贴板时,防止剪贴板的内容发生变化。OpenClipboard 返回 BOOL 值,它说明是否已经成功地打开了剪贴板。如果另一个应用程序没有关闭剪贴板,那么它就不能被打

开。如果每个程序在响应用户的命令时都尽快地、有礼貌地打开然后关闭剪贴板，那么您将永远不会遇到不能打开剪贴板的问题。

但是，在不礼貌程序和抢占式多任务环境中，总会生出一些问题来。即使在程序将某些东西放入剪贴板和用户激活一个 Paste 选项期间，程序并没有丢失输入焦点，但是也不要假定放入的东西仍然在那里，一个后台进程可能在此期间访问过剪贴板。

而且，请留意与消息框有关的一个更微妙的问题：如果不能分配足够的内存来将内容复制到剪贴板，那么您可能希望显示一个消息框。但是，如果这个消息框不是系统模态的，那么用户可以在显示消息框期间切换到另一个应用程序中。您应该或者使消息框是系统模态的，或者在您显示消息框之前关闭剪贴板。

如果在显示一个对话框时将剪贴板保持为打开状态，那么您可能还会遇到其他问题，对话框中的编辑域会使用剪贴板进行文本的剪切和粘贴。

12.1.6 剪贴板和 Unicode

迄今为止，我只讨论了用剪贴板处理 ANSI 文本（每个字符对应一字节）。我们用 CF_TEXT 标识符时的格式。您可能对 CF_OEMTEXT 和 CF_UNICODETEXT 还不熟悉吧。

有一些好消息：在处理首选的文本格式时，只需调用 SetClipboardData 和 GetClipboardData，Windows 将处理剪贴板中的所有文本转换。例如在 Windows NT 中，如果一个程序用 SetClipboardData 来处理 CF_TEXT 剪贴板数据类型，程序也能用 CF_OEMTEXT 调用 GetClipboardData。类似地，剪贴板也能将 CF_OEMTEXT 数据转换为 CF_TEXT。

在 Windows NT 中，转换发生在 CF_UNICODETEXT、CF_TEXT 和 CF_OEMTEXT 之间。程序将用最适合于该程序的文本格式来调用 SetClipboardData 。类似地，程序将用需要的文本格式来调用 GetClipboardData。本书给出的程序在编写时可以带有或不带 UNICODE 标识符。如果您的程序也按此编写，那么当定义了 UNICODE 标识符时，程序将执行调用带有 CF_UNICODETEXT 的 SetClipboardData 和 GetClipboardData 代码，而不是 CF_TEXT。

CLIPTEXT 程序，如程序 12-1 所示，演示了一种可行方法。

程序 12-1 CLIPTEXT 程序

CLIPTEXT.C

```
/*--------------------------------------
   CLIPTEXT.C -- The Clipboard and Text
                 (c) Charles Petzold, 1998
  --------------------------------------*/
```

```
#include <windows.h>
#include "resource.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

#ifdef UNICODE

#define CF_TCHAR CF_UNICODETEXT
TCHAR szDefaultText[] = TEXT ("Default Text-Unicode Version") ;
TCHAR szCaption[]     = TEXT ("Clipboard Text Transfers-Unicode Version") ;
#else

#define CF_TCHAR CF_TEXT
TCHAR szDefaultText[] = TEXT ("Default Text-ANSI Version") ;
TCHAR szCaption[]     = TEXT ("Clipboard Text Transfers-ANSI Version") ;

#endif

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("ClipText") ;
     HACCEL       hAccel ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = szAppName ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, szCaption,
                          WS_OVERLAPPEDWINDOW,
```

```
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     hAccel = LoadAccelerators (hInstance, szAppName) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          if (!TranslateAccelerator (hwnd, hAccel, &msg))
          {
               TranslateMessage (&msg) ;
               DispatchMessage (&msg) ;
          }
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static PTSTR  pText ;
     BOOL          bEnable ;
     HGLOBAL       hGlobal ;
     HDC           hdc ;
     PTSTR         pGlobal ;
     PAINTSTRUCT   ps ;
     RECT          rect ;

     switch (message)
     {
     case WM_CREATE:
          SendMessage (hwnd, WM_COMMAND, IDM_EDIT_RESET, 0) ;
          return 0 ;

     case WM_INITMENUPOPUP:
          EnableMenuItem ((HMENU) wParam, IDM_EDIT_PASTE,
               IsClipboardFormatAvailable (CF_TCHAR) ? MF_ENABLED : MF_GRAYED) ;

          bEnable = pText ? MF_ENABLED : MF_GRAYED ;

          EnableMenuItem ((HMENU) wParam, IDM_EDIT_CUT,    bEnable) ;
          EnableMenuItem ((HMENU) wParam, IDM_EDIT_COPY,   bEnable) ;
          EnableMenuItem ((HMENU) wParam, IDM_EDIT_CLEAR,  bEnable) ;
          break ;
```

```
case WM_COMMAND:
    switch (LOWORD (wParam))
    {
    case IDM_EDIT_PASTE:
        OpenClipboard (hwnd) ;

        if (hGlobal = GetClipboardData (CF_TCHAR))
        {
                 pGlobal = GlobalLock (hGlobal) ;
             if (pText)
             {
                  free (pText) ;
                  pText = NULL ;
             }
             pText = malloc (GlobalSize (hGlobal)) ;
             lstrcpy (pText, pGlobal) ;
             InvalidateRect (hwnd, NULL, TRUE) ;
        }
        CloseClipboard () ;
        return 0 ;

    case IDM_EDIT_CUT:
    case IDM_EDIT_COPY:
        if (!pText)
            return 0 ;

        hGlobal = GlobalAlloc (GHND | GMEM_SHARE,
                               (lstrlen (pText) + 1) * sizeof (TCHAR)) ;
        pGlobal = GlobalLock (hGlobal) ;
        lstrcpy (pGlobal, pText) ;
        GlobalUnlock (hGlobal) ;

        OpenClipboard (hwnd) ;
        EmptyClipboard () ;
        SetClipboardData (CF_TCHAR, hGlobal) ;
        CloseClipboard () ;

        if (LOWORD (wParam) == IDM_EDIT_COPY)
            return 0 ;
                                  // fall through for IDM_EDIT_CUT
    case IDM_EDIT_CLEAR:
        if (pText)
        {
             free (pText) ;
             pText = NULL ;
        }
        InvalidateRect (hwnd, NULL, TRUE) ;
```

```
                    return 0 ;

               case IDM_EDIT_RESET:
                    if (pText)
                    {
                         free (pText) ;
                         pText = NULL ;
                    }
                    pText = malloc ((lstrlen (szDefaultText) + 1) * sizeof (TCHAR)) ;
                    lstrcpy (pText, szDefaultText) ;
                    InvalidateRect (hwnd, NULL, TRUE) ;
                    return 0 ;
               }
               break ;

          case WM_PAINT:
               hdc = BeginPaint (hwnd, &ps) ;

               GetClientRect (hwnd, &rect) ;

               if (pText != NULL)
                    DrawText (hdc, pText, -1, &rect, DT_EXPANDTABS | DT_WORDBREAK) ;

               EndPaint (hwnd, &ps) ;
               return 0 ;

          case WM_DESTROY:
               if (pText)
                    free (pText) ;

               PostQuitMessage (0) ;
               return 0 ;
          }
          return DefWindowProc (hwnd, message, wParam, lParam) ;
     }
```

CLIPTEXT.RC（摘录）

```
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
// Menu

CLIPTEXT MENU DISCARDABLE
BEGIN
    POPUP "&Edit"
```

```
    BEGIN
        MENUITEM "Cu&t \tCtrl+X",           IDM_EDIT_CUT
        MENUITEM "&Copy \tCtrl+C",          IDM_EDIT_COPY
        MENUITEM "&Paste \tCtrl+V",         IDM_EDIT_PASTE

        MENUITEM "De&lete \tDel",           IDM_EDIT_CLEAR
        MENUITEM SEPARATOR
        MENUITEM "&Reset",                  IDM_EDIT_RESET
    END
END

/////////////////////////////////////////////////////////////////////////////
// Accelerator

CLIPTEXT ACCELERATORS DISCARDABLE
BEGIN
    "C",            IDM_EDIT_COPY,          VIRTKEY, CONTROL, NOINVERT
    "V",            IDM_EDIT_PASTE,         VIRTKEY, CONTROL, NOINVERT
    VK_DELETE,      IDM_EDIT_CLEAR,         VIRTKEY, NOINVERT
    "X",            IDM_EDIT_CUT,           VIRTKEY, CONTROL, NOINVERT
END
```

RESOURCE.H(摘录)

```
// Microsoft Developer Studio generated include file.
// Used by ClipText.rc

#define IDM_EDIT_CUT                40001
#define IDM_EDIT_COPY               40002
#define IDM_EDIT_PASTE              40003
#define IDM_EDIT_CLEAR              40004
#define IDM_EDIT_RESET              40005
```

这是在 Windows NT 下运行 Unicode 版和 ANSI 版程序的概念,而且可以看到,剪贴板是如何在两种字符集之间转换的。注意 CLIPTEXT.C 顶部的 #ifdef 语句。如果定义了 UNICODE 标识符,那么 CF_TCHAR(我命名的一种常用的剪贴板格式)就等于 CF_UNICODETEXT;否则,它就等于 CF_TEXT。程序后面调用的 IsClipboardFormatAvailable、GetClipboardData 和 SetClipboardData 函数都使用 CF_TCHAR 名来指定数据类型。

在程序的开始部分(以及从 Edit 菜单中选择 Reset 选项时),静态变量 pText 包含一个指针,在 Unicode 版的程序中,指针指向 Unicode 串 Default Text-Unicode version;在非 Unicode 版的程序中,指针指向 Default Text-ANSI version。您可以用 Cut 或 Copy 命令将文本串传送到剪贴板,用 Cut 或 Delete 命令从程序中删除串。Paste 命令将剪贴板中的文本内容复制到 pText。在 WM_PAINT 消息期间,pText 将串显示在程序的客户区。

如果先在 Unicode 版的 CLIPTEXT 中选择了 Copy 命令，然后在非 Unicode 版中选择 Paste 命令，那么您就能看到文本已经从 Unicode 转换成了 ANSI。类似地，如果您执行相反的操作，那么文本就会从 ANSI 转换成 Unicode。

12.2 复杂的剪贴板用法

我们已经看到，在将数据准备好之后，从剪贴板传输数据时需要 4 个调用：

```
OpenClipboard (hwnd) ;
EmptyClipboard () ;
SetClipboardData (iFormat, hGlobal) ;
CloseClipboard () ;
```

访问这些数据需要 3 个调用：

```
OpenClipboard (hwnd) ;
hGlobal = GetClipboardData (iFormat) ;
[other program lines]
CloseClipboard () ;
```

在 GetClipboardData 和 CloseClipboard 调用之间，可以复制剪贴板数据或以其他方式来使用它。很多应用程序都需要采用这种方法，但也可以用更复杂的方式来使用剪贴板。

12.2.1 利用多个数据项

当打开剪贴板并把数据传送给它时，必须先调用 EmptyClipboard，通知 Windows 释放或删除剪贴板上的内容。不能在现有的剪贴板内容中附加其他东西。所以，从这种意义上说，剪贴板每次只能保留一个数据项。

但是，可以在 EmptyClipboard 和 CloseClipboard 调用之间多次调用 SetClipboardData，每次都使用不同的剪贴板格式。例如，如果想在剪贴板中存储一个很短的文本字符串，可以把这个文本写入元文件，也可以把这个文本写入位图。把位图选进内存设备描述表中，把这个字符串写进位图中。利用这种方法可以使字符串不仅能为从剪贴板上读取文本的程序所使用，也可以为从剪贴板上读取位图和元文件的程序所使用。当然，这些程序并不能知道元文件或位图实际上包含了一个字符串。

如果想把一些句柄写到剪贴板上，对每个句柄均可以调用 SetClipboardData：

```
OpenClipboard (hwnd) ;
EmptyClipboard () ;
SetClipboardData (CF_TEXT, hGlobalText) ;
SetClipboardData (CF_BITMAP, hBitmap) ;
SetClipboardData (CF_METAFILEPICT, hGlobalMFP) ;
```

```
CloseClipboard () ;
```

当这三种格式的数据同时位于剪贴板上时，用 CF_TEXT、CF_BITMAP 或 CF_METAFILEPICT 参数调用 IsClipboardFormatAvailable 将返回 TRUE。通过下列调用程序可以访问这些句柄：

```
hGlobalText = GetClipboardData (CF_TEXT) ;
```

或

```
hBitmap = GetClipboardData (CF_BITMAP) ;
```

或

```
hGlobalMFP = GetClipboardData (CF_METAFILEPICT) ;
```

下一次程序调用 EmptyClipboard 时，Windows 将释放或删除剪贴板上保留的所有三个句柄。

在将不同的文本格式、不同的位图格式或者不同的元文件格式添加到剪贴板时，不要使用这种技术。只使用一种文本格式、一种位图格式，以及一种元文件格式。就像我所说的那样，Windows 将在 CF_TEXT、CF_OEMTEXT 和 CF_UNICODETEXT 之间转换，也可以在 CF_BITMAP 和 CF_DIB 之间，以及在 CF_METAFILEPICT 和 CF_ENHMETAFILE 之间进行转换。

通过先打开剪贴板，然后调用 EnumClipboardFormats，程序可以确定剪贴板存储的所有格式。开始时设置变量 iFormat 为 0：

```
iFormat = 0 ;
OpenClipboard (hwnd) ;
```

现在从 0 值开始逐次进行连续的 EnumClipboardFormats 调用。函数将为当前在剪贴板中的每种格式返回一个正的 iFormat 值。当函数返回 0 时，表示完成：

```
while (iFormat = EnumClipboardFormats (iFormat))
{
    [logic for each iFormat value]
}
CloseClipboard () ;
```

您可以通过下面的调用来获取当前在剪贴板中的不同格式的个数：

```
iCount = CountClipboardFormats () ;
```

12.2.2 延迟生成

当把数据放入剪贴板中时，一般来说要制作一份数据的副本，并将包含这份副本的内存块句柄传给剪贴板。对非常大的数据项来说，这种方法会浪费内存空间。如果用户不想把数据粘贴到另一个程序里，那么在被其他内容取代之前，它将一直占据着内存空间。

通过使用一种叫做"延迟生成"的技术可以避免这个问题。实际上，直到另一个程序需要数据时，程序才提供这份数据。为此，不将数据句柄传送给 Windows，而是在 SetClipboardData 调用中使用 NULL：

```
OpenClipboard (hwnd) ;
EmptyClipboard () ;
SetClipboardData (iFormat, NULL) ;
CloseClipboard () ;
```

可以有多个使用不同 iFormat 值的 SetClipboardData 调用，对其中某些调用可使用 NULL 值。而对其他一些则使用实际的句柄值。

前面的过程比较简单，以下过程就要稍微复杂一些了。当另一个程序调用 GetClipboardData 时，Windows 将检查那种格式的句柄是否为 NULL。如果是，Windows 将给"剪贴板所有者"（您的程序）发送一个消息，请求数据的实际句柄，这时您的程序必须提供这个句柄。

更具体地说，"剪贴板所有者"是将数据放入剪贴板的最后一个窗口。当一个程序调用 OpenClipboard 时，Windows 保存调用这个函数时所用的窗口句柄，这个句柄标识打开剪贴板的窗口。一旦收到一个 EmptyClipboard 调用，Windows 就使这个窗口作为新的剪贴板所有者。

使用延迟生成技术的程序在它的窗口过程中必须处理三个消息：WM_RENDERFORMAT、WM_RENDERALLFORMATS 和 WM_DESTROYCLIPBOARD。当另一个程序调用 GetClipboardData 时，Windows 给窗口过程发送一个 WM_RENDERFORMAT 消息，wParam 的值是所要求的格式。在处理 WM_RENDERFORMAT 消息时，不要打开或清空剪贴板。为 wParam 所指定的格式创建一个全局内存块，把数据传给它，并用正确的格式和相应句柄调用 SetClipboardData。很明显，为了在处理 WM_RENDERFORMAT 时正确地构造出此数据，需要在程序中保留这些信息。当另一个程序调用 EmptyClipboard 时，Windows 给您的程序发送一个 WM_DESTROYCLIPBOARD 消息告诉您不再需要构造剪贴板数据的信息。您的程序不再是剪贴板的所有者。

如果程序在它自己仍然是剪贴板所有者的时候就要终止运行，并且剪贴板上仍然包含着该程序用 SetClipboardData 设置的 NULL 数据句柄，它将收到 WM_RENDERALL-

FORMATS 消息。这时，应该打开剪贴板，清空它，把数据加载到内存块中，并为每种格式调用 SetClipboardData，然后关闭剪贴板。WM_RENDERALLFORMATS 消息是窗口过程最后收到的消息之一。它后面跟有 WM_DESTROYCLIPBOARD 消息（由于已经生成了所有数据），然后是正常的 WM_DESTROY 消息。

如果您的程序只能向剪贴板传输一种格式的数据（例如文本），那么您可以把 WM_RENDERALLFORMATS 和 WM_RENDERFORMAT 处理结合在一起。这些代码与下面的代码相似：

```
case WM_RENDERALLFORMATS :
     OpenClipboard (hwnd) ;
     EmptyClipboard () ;
                              // fall through
case WM_RENDERFORMAT :
     [put text into global memory block]
     SetClipboardData (CF_TEXT, hGlobal) ;

     if (message == WM_RENDERALLFORMATS)
          CloseClipboard () ;
     return 0 ;
```

如果您的程序使用几种剪贴板格式，那么您可能想为 wParam 所要求的格式处理 WM_RENDERFORMAT。除非程序难于保存构造数据所需的信息，否则不需要处理 WM_DESTROYCLIPBOARD 消息。

12.2.3 私有数据格式

到目前为止，我们仅处理了 Windows 定义的标准剪贴板数据格式。但是，您可能想用剪贴板来存储"私有数据格式"。许多字处理程序使用这种技术来存储包含着字体和格式化信息的文本。

初看起来，这个概念似乎是没有意义的。如果剪贴板的作用是在应用程序之间传送数据，那么，为什么剪贴板中要含有只有一个应用程序才能理解的数据呢？答案很简单：剪贴板允许在同一个程序的内部（或者可能在一个程序中的不同实例之间）传送数据，很明显，这些实例能理解它们自己的私有数据格式。

有几种使用私有数据格式的方法。最简单的方法用到一种表面上是标准剪贴板格式（文本、位图或元文件）的、只对您的程序才有意义的数据。这种情况下，在 SetClipboardData 和 GetClipboardData 调用中可使用下列 wFormat 值：CF_DSPTEXT、CF_DSPBITMAP、CF_DSPMETAFILEPICT 或 CF_DSPENHMETAFILE。（字母 DSP 代表"显示"。）这些格式允许 Windows 按文本、位图或元文件来浏览或显示数据。但是，另一个使用常规的

CF_TEXT、CF_BITMAP、CF_DIB、CF_METAFILEPICT 或 CF_ENHMETAFILE 格式调用 GetClipboardData 的程序不能获取这个数据。

如果用其中一种格式把数据放入剪贴板中，则必须使用同样的格式读出数据。但是，如何知道数据是来自程序的另一个实例，还是来自使用其中某种数据格式的另一个程序呢？这里有一种方法，可以通过下列调用首先获得剪贴板所有者：

```
hwndClipOwner = GetClipboardOwner () ;
```

然后可以得到此窗口句柄的窗口类名称：

```
TCHAR szClassName [32] ;
[other program lines]
GetClassName (hwndClipOwner, szClassName, 32) ;
```

如果类名与程序名相同，那么数据是由程序的另一个实例传送到剪贴板中的。

使用私有数据格式的第二种方法涉及到 CF_OWNERDISPLAY 标志。SetClipboard-Data 的全局句柄是 NULL：

```
SetClipboardData (CF_OWNERDISPLAY, NULL) ;
```

这是某些字处理器在 Windows 的剪贴板查看器的客户区域中显示格式化文本时所采用的方法。很明显，剪贴板查看器不知道如何显示这种格式化文本。当一个字处理器指定 CF_OWNERDISPLAY 格式时，它也就承担起在剪贴板查看器的客户区域中绘图的责任。

由于全局内存句柄为 NULL，所以用 CF_OWNERDISPLAY 格式（剪贴板所有者）调用 SetClipboardData 的程序必须处理由 Windows 发往剪贴板所有者的延迟生成消息，以及 5 个附加消息。这 5 个消息是由剪贴板查看器发送到剪贴板所有者的：

- **WM_ASKCBFORMATNAME** 剪贴板查看器把这个消息发送到剪贴板所有者，以得到数据格式名称。lParam 参数是指向缓冲区的指针，wParam 是该缓冲区能容纳的最大字符数目。剪贴板所有者必须把剪贴板数据格式的名称复制到该缓冲区中。
- **WM_SIZECLIPBOARD** 这个消息通知剪贴板所有者，剪贴板查看器的客户区域的大小已发生了变化。wParam 参数是剪贴板查看器的句柄，lParam 是指向包含新尺寸的 RECT 结构的指针。如果 RECT 结构中都是 0，则剪贴板查看器退出或最小化。尽管 Windows 的剪贴板查看器只允许它自己的一个实例运行，但其他剪贴板查看器也能把这个消息发送给剪贴板所有者。对付多个剪贴板查看器并非不可能（假定 wParam 标识特定的查看器），但剪贴板所有者处理起来也不容易。
- **WM_PAINTCLIPBOARD** 这个消息通知剪贴板所有者修改剪贴板查看器的客户

区。同时,wParam 是剪贴板查看器窗口的句柄,lParam 是指向 PAINTSTRUCT 结构的全局指针。剪贴板所有者可以从此结构的 hdc 域中得到剪贴板查看器设备描述表句柄。

- **WM_HSCROLLCLIPBOARD 和 WM_VSCROLLCLIPBOARD** 这两个消息通知剪贴板所有者,用户已经滚动了剪贴板查看器的滚动条。wParam 参数是剪贴板查看器窗口的句柄,lParam 的低位字是滚动请求,并且,如果低位字是 SB_THUMBPOSITION,那么 lParam 的高位字是滑块位置。

处理这些消息比较麻烦,看来并不值得这样做。但是,这种处理对用户来说是有益的;当从字处理程序把文本复制到剪贴板时,用户在剪贴板查看器的客户区中看见文本还保持着格式时,心里会舒坦些。

使用私有剪贴板数据格式的第三种方法是注册自己的剪贴板格式名。向 Windows 提供格式名,Windows 给程序提供一个序号,它可以用作 SetClipboardData 和 GetClipboardData 的格式参数。一般来说,采用这种方法的程序也要以一种标准格式把数据复制到剪贴板。这种方法允许剪贴板查看器在它的客户区中显示数据(没有与 CF_OWNERDISPLAY 相关的冲突),并且允许其他程序从剪贴板上复制数据。

例如,假定我们已经编写了一个以位图格式、元文件格式和自己的已注册的剪贴板格式把数据复制到剪贴板中的矢量绘图程序。剪贴板查看器将显示元文件或者位图,其他从剪贴板上读取位图和元文件的程序将获得这几种格式。但是,当我们的矢量绘图程序需要从剪贴板上读取数据时,它会按照自己已注册的格式复制数据,这是因为这种格式可能包含着比位图文件或者元文件更多的信息。

程序通过下面的调用来注册一个新的剪贴板格式:

```
iFormat = RegisterClipboardFormat (szFormatName) ;
```

iFormat 的值介于 0xC000 和 0xFFFF 之间。剪贴板查看器(或一个通过调用 EnumClipboardFormats 获取当前所有剪贴板数据格式的程序)可以获取这种数据格式的 ASCII 名,这是通过下面调用实现的:

```
GetClipboardFormatName (iFormat, psBuffer, iMaxCount) ;
```

Windows 将多达 iMaxCount 个字符复制到 psBuffer 中。

使用这种方法把数据复制到剪贴板中的程序员可能需要公开数据格式名称和实际的数据格式。如果这个程序流行起来,那么其他程序就会以这种格式从剪贴板复制数据。

12.3 实现剪贴板查看器

监视剪贴板内容变化的程序称为"剪贴板查看器"。您可以在 Windows 中得到一个剪贴

板查看器，但是您也可以编写自己的剪贴板查看器程序。剪贴板查看器通过传递到查看器窗口过程的消息来监视剪贴板内容的变化。

12.3.1 剪贴板查看器链

任意数量的剪贴板查看器应用程序都可以同时在 Windows 下运行，它们都可以监视剪贴板内容的变化情况。但是，从 Windows 的角度来看，只存在一个剪贴板查看器，我们称之为“当前剪贴板查看器”。Windows 只保留一个识别当前剪贴板查看器的窗口句柄，并且当剪贴板的内容发生变化时，只把消息发送到那个窗口中。

剪贴板查看器应用程序有必要加入“剪贴板查看链”，以便所有运行的剪贴板查看器都可以收到 Windows 发送给当前剪贴板查看器的消息。当一个程序将自己注册为一个剪贴板查看器时，它就成为当前剪贴板查看器。Windows 把先前的当前查看器的窗口句柄交给这个程序，然后该程序保存这个句柄。当此程序收到一个剪贴板查看器消息时，它把这个消息发送给剪贴板链中下一个程序的窗口过程。

12.3.2 剪贴板查看器的函数和消息

程序通过调用 SetClipboardViewer 函数可以成为剪贴板查看器链的一部分。如果程序的主要目的是作为剪贴板查看器来提供服务，那么这个程序在处理 WM_CREATE 消息期间可以调用这个函数，该函数返回前一个当前剪贴板查看器的窗口句柄。程序应该把这个句柄保存在静态变量中：

```
static HWND hwndNextViewer ;
[other program lines]
case WM_CREATE :
     [other program lines]
     hwndNextViewer = SetClipboardViewer (hwnd) ;
```

如果在 Windows 的一次运行期间您的程序是成为剪贴板查看器的第一个程序，那么 hwndNextViewer 将为 NULL。

不管剪贴板中的内容怎样变化，Windows 都将把 WM_DRAWCLIPBOARD 消息发送给当前剪贴板查看器(最近注册为剪贴板查看器的窗口)。剪贴板查看器链中的每个程序都应该用 SendMessage 把这个消息发送到下一个剪贴板查看器。查看器链中的最后一个程序(第一个将自己注册为剪贴板查看器的窗口)保存的 hwndNextViewer 为 NULL。如果 hwndNextViewer 为 NULL，那么程序只简单地返回，而不向其他程序发送任何消息(不要把 WM_DRAWCLIPBOARD 消息和 WM_PAINTCLIPBOARD 消息混淆起来。WM_PAINTCLIPBOARD 是由剪贴板查看器发送给使用 CF_OWNERDISPLAY 剪贴板数据格式的程序的，而 WM_DRAWCLIPBOARD 消息是由 Windows 发往当前剪贴板查看器的)。

处理 WM_DRAWCLIPBOARD 消息的最简单的方法是将消息发送给下一个剪贴板查看器(除非 hwndNextViewer 为 NULL),并使窗口的客户区域无效:

```
case WM_DRAWCLIPBOARD :
    if (hwndNextViewer)
        SendMessage (hwndNextViewer, message, wParam, lParam) ;

    InvalidateRect (hwnd, NULL, TRUE) ;
    return 0 ;
```

在处理 WM_PAINT 消息期间,通过使用常规的 OpenClipboard、GetClipboardData 和 CloseClipboard 调用可以读取剪贴板的内容。

当某个程序想从剪贴板查看器链中删除它自己时,它必须调用 ChangeClipboardChain。这个函数接收脱离查看器链的程序的窗口句柄和下一个剪贴板查看器的窗口句柄:

```
ChangeClipboardChain (hwnd, hwndNextViewer) ;
```

当程序调用 ChangeClipboardChain 时,Windows 将 WM_CHANGECBCHAIN 消息发送给当前剪贴板查看器。wParam 参数是从链中删除它自己的那个查看器的窗口句柄(ChangeClipboardChain 的第一个参数),lParam 是从链中删除自己后的下一个剪贴板查看器的窗口句柄(ChangeClipboardChain 的第二个参数)。

当程序接收到 WM_CHANGECBCHAIN 消息时,必须检查 wParam 是否等于已经保存的 hwndNextViewer 的值。如果是这样,程序必须设置 hwndNextViewer 为 lParam。这项工作保证将来的 WM_DRAWCLIPBOARD 消息不会发送给从剪贴板查看器链中删除了自己的窗口。如果 wParam 不等于 hwndNextViewer ,并且 hwndNextViewer 不为 NULL,则把消息送到下一个剪贴板查看器:

```
case WM_CHANGECBCHAIN :
    if ((HWND) wParam == hwndNextViewer)
        hwndNextViewer = (HWND) lParam ;

    else if (hwndNextViewer)
        SendMessage (hwndNextViewer, message, wParam, lParam) ;
    return 0 ;
```

不一定要使用 else if 语句,它只用于保证 hwndNextViewer 为非 NULL 的值。hwndNextViewer 的值为 NULL 时,执行这段代码的程序就是链中最后一个查看器,而这是不可能的。

当程序快结束时,如果它仍然在剪贴板查看器链中,则必须从链中删除它。您可以在处理 WM_DESTROY 消息时调用 ChangeClipboardChain 来完成这项工作。

```
case WM_DESTROY :
    ChangeClipboardChain (hwnd, hwndNextViewer) ;
    PostQuitMessage (0) ;
    return 0 ;
```

Windows 还有一个允许程序获得第一个剪贴板查看器窗口句柄的函数:

```
hwndViewer = GetClipboardViewer () ;
```

一般来说不需要这个函数。如果没有当前剪贴板查看器,则返回值为 NULL。

下面是一个说明剪贴板查看器链如何工作的例子。当 Windows 刚启动时,当前剪贴板查看器是 NULL:

剪贴板查看器: NULL

具有 hwnd1 窗口句柄的程序调用 SetClipboardViewer。这个函数返回的 NULL 成为这个程序中的 hwndNextViewer 值:

当前剪贴板查看器: hwnd1
hwnd1 的下一个查看器: NULL

第二个具有 hwnd2 窗口句柄的程序调用 SetClipboardViewer,并返回 hwnd1:

当前的剪贴板查看器: hwnd2
hwnd2 的下一个查看器: hwnd1
hwnd1 的下一个查看器: NULL

每三个程序(hwnd3)和第四个程序(hwnd4)也调用 SetClipboardViewer,并且返回 hwnd2 和 hwnd3:

当前的剪贴板查看器: hwnd4
hwnd4 的下一个查看器: hwnd3
hwnd3 的下一个查看器: hwnd2
hwnd2 的下一个查看器: hwnd1
hwnd1 的下一个查看器: NULL

当剪贴板的内容发生变化时,Windows 将 WM_DRAWCLIPBOARD 消息发送给 hwnd4,hwnd4 将消息发送给 hwnd3,hwnd3 将消息发送给 hwnd2,hwnd2 将消息发送给 hwnd1,hwnd1 返回。

现在 hwnd2 决定通过下列调用从链中删除自己:

```
ChangeClipboardChain (hwnd2, hwnd1) ;
```

Windows 将 wParam 等于 hwnd2、lParam 等于 hwnd1 的 WM_CHANGECBCHAIN 消息发

送给 hwnd4。由于 hwnd4 的下一个查看器是 hwnd3,所以 hwnd4 把这个消息传给 hwnd3。现在 hwnd3 注意到 wParam 等于它的下一个查看器(hwnd2),所以将下一个查看器设置为 lParam (hwnd1)并且返回。这样工作就完成了。现在剪贴板查看器链如下:

当前剪贴板查看器:	hwnd4
hwnd4 的下一个查看器:	hwnd3
hwnd3 的下一个查看器:	hwnd1
hwnd1 的下一个查看器:	NULL

12.3.3 一个简单的剪贴板查看器

剪贴板查看器不一定要像 Windows 所提供的那样完善,例如,剪贴板查看器可以只显示一种剪贴板数据格式。程序 12-2 中所示的 CLIPVIEW 程序是一种只能显示 CF_TEXT 格式的剪贴板查看器。

程序 12-2 CLIPVIEW 程序

CLIPVIEW.C

```
/*----------------------------------------
     CLIPVIEW.C——Simple Clipboard Viewer
                 (c) Charles Petzold, 1998
  ----------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("ClipView") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style          = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc    = WndProc ;
     wndclass.cbClsExtra     = 0 ;
     wndclass.cbWndExtra     = 0 ;
     wndclass.hInstance      = hInstance ;
     wndclass.hIcon          = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground  = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName   = NULL ;
```

```
     wndclass.lpszClassName  = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName,
                          TEXT ("Simple Clipboard Viewer (Text Only)"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static HWND hwndNextViewer ;
     HGLOBAL     hGlobal ;
     HDC         hdc ;
     PTSTR       pGlobal ;
     PAINTSTRUCT ps ;
     RECT        rect ;

     switch (message)
     {
     case WM_CREATE:
          hwndNextViewer = SetClipboardViewer (hwnd) ;
          return 0 ;

     case WM_CHANGECBCHAIN:
          if ((HWND) wParam == hwndNextViewer)
               hwndNextViewer = (HWND) lParam ;

          else if (hwndNextViewer)
               SendMessage (hwndNextViewer, message, wParam, lParam) ;
```

```
            return 0 ;

        case WM_DRAWCLIPBOARD:
            if (hwndNextViewer)
                SendMessage (hwndNextViewer, message, wParam, lParam) ;

            InvalidateRect (hwnd, NULL, TRUE) ;
            return 0 ;

        case WM_PAINT:
            hdc = BeginPaint (hwnd, &ps) ;
            GetClientRect (hwnd, &rect) ;
            OpenClipboard (hwnd) ;

#ifdef UNICODE
            hGlobal = GetClipboardData (CF_UNICODETEXT) ;
#else
             hGlobal = GetClipboardData (CF_TEXT) ;
#endif

             if (hGlobal != NULL)
             {
                  pGlobal = (PTSTR) GlobalLock (hGlobal) ;
                  DrawText (hdc, pGlobal, -1, &rect, DT_EXPANDTABS) ;
                  GlobalUnlock (hGlobal) ;
             }

             CloseClipboard () ;
             EndPaint (hwnd, &ps) ;
             return 0 ;

         case WM_DESTROY:
             ChangeClipboardChain (hwnd, hwndNextViewer) ;
             PostQuitMessage (0) ;
             return 0 ;
         }
         return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

CLIPVIEW 按上面所讨论的方法来处理 WM_CREATE、WM_CHANGECBCHAIN、WM_DRAWCLIPBOARD 和 WM_DESTROY 消息。WM_PAINT 消息处理打开剪贴板，并用 CF_TEXT 格式调用 GetClipboardData。如果函数返回一个全局内存句柄，那么 CLIPVIEW 将锁定它，并用 DrawText 在客户区显示文本。

处理标准格式（如 Windows 提供的那个剪贴板一样）以外的数据格式的剪贴板查看

器还需要完成其他一些工作，比如显示剪贴板中当前所有数据格式的名称。用户可以通过调用 EnumClipboardFormats，并使用 GetClipboardFormatName 得到非标准数据格式名称来完成这项工作。使用 CF_OWNERDISPLAY 数据格式的剪贴板查看器必须把下面 4 个消息传送给剪贴板数据的拥有者，以显示该数据：

```
WM_PAINTCLIPBOARD WM_VSCROLLCLIPBOARD
WM_SIZECLIPBOARD   WM_HSCROLLCLIPBOARD
```

如果您想编写这样的剪贴板查看器，那么必须使用 GetClipboardOwner 来获得剪贴板所有者的窗口句柄，并且在需要修改剪贴板的客户区时，必须将这些消息发送给该窗口。

Document generated by Anna’s Archive around 2023-2024 as part of the DuXiu collection (https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):
{
  "filename": "MTExNTU4MDQuemlw",
  "filename_decoded": "11155804.zip",
  "filesize": 52362485,
  "md5": "036374c06cab537b6606d43e488dc6b0",
  "header_md5": "088030e8210845a8ce71cf8631ab98c8",
  "sha1": "e3cbfc9ff967a3510f59126551a65d0537f1e2b1",
  "sha256": "142af72526d102d19a8ca71adbba99c8cde8ea6f2825f31463211f97940d3a36",
  "crc32": 2084549502,
  "zip_password": "",
  "uncompressed_size": 57646125,
  "pdg_dir_name": "11155804_Windows\u2502\u2560\u2568\u2265\u2554\u03a6\u255d\u255e\u2561\u250c5\u2591\u00b5\u2554\u2567\u2593\u00df",
  "pdg_main_pages_found": 556,
  "pdg_main_pages_max": 556,
  "total_pages": 572,
  "total_pixels": 3347084480,
  "pdf_generation_missing_pages": false
}